НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»

УДК _____

«До захисту допущено»

Завідувач кафедри

Стіренко С.Г.
(підпис) (ініціали, прізвище)

“____” _____ 2020 р.

Магістерська дисертація

зі спеціальності: *121. Інженерія програмного забезпечення*
(код та назва напряму підготовки або спеціальності)

Спеціалізація: *121.Програмне забезпечення високопродуктивних комп’ютерних систем та мереж*

на тему: Метод оптимізації паралельних обчислень в розподілених гетерогенних комп’ютерних системах на основі багаторівневого балансування навантаження

Виконав: студент 6 курсу, групи ІП-83МН
(шифр групи)

Демчик Валерій Валентинович
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник доц., к.т.н., доц. Корочкін О. В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Розділи 1-3 доц., к.т.н., доц. Русанова О. В.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Консультант Нормоконтроль проф., д.т.н., проф. Жабін В. І.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Освітньо-кваліфікаційний ступінь *магістр*
(назва ОКР)

Спеціальність *121. Інженерія програмного забезпечення*
(код і назва)

Спеціалізація *121. Програмне забезпечення високопродуктивних комп'ютерних систем та мереж*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
« » 2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Демчику Валерію Валентиновичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Метод оптимізації паралельних обчислень в розподілених гетерогенних комп'ютерних системах на основі багаторівневого балансування навантаження
Науковий керівник дисертації Корочкін О. В. к.т.н., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 24 березня 2020 року № 910-с

2. Строк подання студентом дисертації

3. Об'єкт дослідження паралельні та розподілені обчислення, високонавантажені обчислення

4. Предмет дослідження методи та засоби організації паралельних обчислень в розподілених гетерогенних комп'ютерних системах (РГКС), методи планування обчислень та балансування навантаження в РГКС.

5. Перелік завдань, які потрібно розробити:

1. Виконати аналіз сучасних підходів до організації паралельних обчислень в РГКС, а також основні методи оптимізації таких обчислень.

2. Виділити основні проблеми які виникають при організації РГКС та паралельних обчислень в них, дослідити та систематизувати існуючі пропозиції їх вирішення.

3. Запропонувати методи та підходи до вирішення виявлених проблем, які дозволять підвищити ефективність паралельних обчислень в РГКС.

4. Провести експериментальні дослідження ефективності запропонованих методів та підходів шляхом розробки та тестування пакету програм для реальних РГКС

5. Провести порівняльний аналіз створених програмних комплексів для РГКС із вже існуючими реалізаціями альтернативних підходів за показниками максимально досягнутих показників прискорення та ефективності.

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1,2,3	доц., к.т.н., доц. Русанова О. В.		
Нормоконтроль	проф., д.т.н., проф. Жабін В. І.		

7. Дата видачі завдання 03 лютого 2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1	Аналіз предметної області, огляд літератури	08.03.2020	
2	Виділення недоліків моделей та технологій організації паралельних обчислень в РГКС	08.03.2020	
3	Розробка методів оптимізації паралельних обчислень та їх балансування в РГКС	08.03.2020	
5	Розробка програмних прототипів	27.03.2020	
6	Тестування та аналіз результатів роботи системи	10.04.2020	
7	Оформлення матеріалів дисертації	24.04.2020	

Студент _____ В. В. Демчик
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ О. В. Корочкін
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Структура і обсяг роботи: магістерська дисертація викладена на 128 сторінках, складається зі вступу, 3 розділів, висновку, містить 20 рисунків, 4 таблиці, 13 формул, список використаних джерел із 33 найменувань на 5 сторінках.

Актуальність роботи. Станом на сьогодні спостерігається поява все більшої кількості комп'ютерних систем які мають гетерогенну структуру, засновану на використанні центрального процесора та акселераторів з відмінною від нього архітектурою. В той же час, для достатнього рівня швидкодії складних програмних комплексів стає не достатньо застосування лише паралельних систем, стандартною практикою для високонавантажених обчислень стає їх розгортка в розподілених комп'ютерних системах, для яких паралельні системи є лише складовими елементами. Для всіх розподілених систем одним з ключових факторів, що визначають ефективність роботи системи, є вміле балансування навантаження на всіх рівнях цієї системи. Вирішенню цієї NP-повної задачі наразі присвячено цілий розділ комп'ютерних наук, але вона досі лишається актуальною проблемою, оскільки абсолютно ідеального рішення за прийнятний час в ній знайти наразі не можливо. А в випадку, коли кінцеві вузли розподіленої системи являють собою гетерогенні комп'ютерні системи, то задача ускладняється ще більше, оскільки необхідно провести балансування обчислень між різнорідними елементами із зазвичай значно відмінними між собою характеристиками продуктивності та комунікабельності.

Наразі існує великий набір стандартних методів балансування навантаження в розподілених системах, але більшість із них орієнтовані на конкретний тип задач, лише невелика кількість не прив'язані до конкретних задач. сучасні засоби організації паралельних обчислень в розподілених комп'ютерних системах з акселераторами приймають низький рівень абстракції, що веде до рядку складнощів та обмежень. Вбудовані в них методи

балансування навантаження орієнтовані на гомогенні і здебільшого повнозв'язні системи.

Мета роботи: підвищення ефективності проведення паралельних обчислень в гетерогенних розподілених комп'ютерних системах шляхом оптимального розподілу та балансування навантаження на її складові елементи.

Завдання дослідження:

1. Виконати аналіз сучасних підходів до організації паралельних обчислень в РГКС, а також основні методи оптимізації таких обчислень.
2. Виділити основні проблеми які виникають при організації РГКС та паралельних обчислень в них, дослідити та систематизувати існуючі пропозиції їх вирішення.
3. Запропонувати методи та підходи до вирішення виявлених проблем, які дозволять підвищити ефективність паралельних обчислень в РГКС.
4. Провести експериментальні дослідження ефективності запропонованих методів та підходів шляхом розробки та тестування пакету програм для реальних РГКС.
5. Провести порівняльний аналіз створених програмних комплексів для РГКС із вже існуючими реалізаціями альтернативних підходів за показниками максимально досягнутих показників прискорення та ефективності.

Об'єкт дослідження: паралельні та розподілені обчислення, високопродуктивні обчислення.

Предмет дослідження: методи організації паралельних обчислень в розподілених гетерогенних комп'ютерних системах, методи планування обчислень та балансування навантаження в РГКС.

Методи дослідження: методи статистичного опрацювання даних, теорія паралельних та розподілених обчислень, теорія планування, теорія оптимізації, теорія компіляторів, теорія алгоритмів, теорія графів.

Публікації:

1. «Дослідження ефективності дрібнозернистого паралелізму в багатоядерних комп'ютерних системах», «Вісник НТУУ «КПІ. Інформатика, управління

та обчислювальна техніка: зб. наук. праць», 2020, № 66, с. 56 – 61. Також результати та розвиток цього дослідження були представлені на міжнародних конференціях CSNT-2017, CSNT-2018, ICSFTI-2018 та опубліковані у відповідних збірниках праць даних конференцій.

2. «Neural network acceleration method in the two-component CPU-GPU computer systems», Збірник тез доповідей VI Міжнародної конференції «High Performance Computing» (HPC-UA 2020) [на рев'ю]. Також матеріали цього дослідження були представлені на міжнародних конференціях CSNT-2019, ICSFTI-2019 та опубліковані у відповідних збірниках праць даних конференцій.
3. «Застосування технології WCF для підвищення ефективності обчислень в сучасних розподілених комп'ютерних системах», Збірник тез доповідей XIII Міжнародної науково-технічної конференції «Комп'ютерні системи та мережні технології» (CSNT-2020) [очікує публікації].
4. «Застосування технології WCF для підвищення ефективності паралельних обчислень в хмарних розподілених комп'ютерних системах», Безпека. Відмовостійкість. Інтелект: збірник праць міжнародної науково-практичної конференції ICSFTI2020 [очікує публікації].

Ключові слова: паралельні обчислення, розподілені комп'ютерні системи, гетерогенні комп'ютерні системи, кластерні системи, акселератори, планування обчислень, балансування навантаження, відвантажені обчислення, відкладені обчислення, потоки, ядра, графічні процесори.

ABSTRACT

Structure and scope of master's thesis: Master's thesis is presented on 128 pages, consists of introduction, 3 sections, conclusion, contains 20 drawings, 4 tables, 13 formulas, a bibliography of 33 titles on 5 pages.

Relevance of work. Today, an increasing number of computer systems have a heterogeneous structure based on the use of central processing unit and accelerators with a different architecture. At the same time, it is not enough to use only parallel systems for a sufficient level of performance of complex software systems; it is standard practice for high-load computations to deploy them in distributed computer systems, in which parallel systems are only constituent elements. For all distributed systems, one of the key characteristics of the system performance is the ability to balance workloads at all levels of the system. A whole section of computer science is currently devoted to solving NP-complete problem, but it still remains an actual problem, since it is not possible to find the perfect solution in an acceptable time for now. And if the end nodes of a distributed system are heterogeneous computer systems, the task becomes even more complicated as it is necessary to balance the computations between heterogeneous elements with typically significantly different performance and communication characteristics.

Currently, there is a large set of standard load balancing methods in distributed systems, but most of them are focused on specific types of tasks, and only a small number are not tied to them. Modern parallel computing tools in distributed computer systems with accelerators accept a low level of abstraction, leading to a number of difficulties and limitations. The load balancing techniques built into them are oriented on homogeneous and fully connected systems.

Purpose of work: increasing the efficiency of parallel computing in distributed heterogeneous computer systems by optimally distributing and balancing the load on its elements.

Research tasks:

1. To analyze modern approaches to the organization of parallel calculations in DHCS, as well as the basic methods of optimization of such computations;

2. To identify the main problems that arise in the organization of DHCS and parallel calculations in them, to investigate and systematize existing proposals for their solution;
3. To highlight the main problems and disadvantages of existing approaches and technologies, propose own solutions of the identified problems and ideas for improving the work of DHCS;
4. Conduct experimental studies of the effectiveness of the proposed methods and approaches by developing and testing a software systems for real DHCS;
5. To carry out a comparative analysis of the created software systems for DHCS with already existing implementations of alternative approaches based on the achieved acceleration and efficiency.

Object of research: parallel and distributed computing, high-performance computing.

Subject of research: methods of organization of parallel calculations in distributed heterogeneous computer systems, methods of calculations planning and load balancing in DHCS.

Research methods: methods of statistical data processing, theory of parallel and distributed calculations, theory of planning, theory of optimization, theory of compilers, theory of algorithms, theory of graphs.

Publications:

1. “Doslidzhennia efektyvnosti dribnozernystoho paralelizmu v bahatoiadernykh kompiuternykh systemakh” [The investigation of effectiveness in using of fine-grained parallelism in multicore computer systems], Visnyk NTUU “KPI”. Informatyka, upravlinnia ta obchysliuvalna tekhnika: zbirnyk naukovykh prats [Herald of NTUU “KPI”. Information technology, management and computing technics: a collection of scientific papers], vol. 66, pp. 56 – 61., in press. Also, the results of this study were presented at international conferences CSNT-2017, CSNT-2018, ICSFTI-2018 and published in the relevant proceedings of these conferences.
2. «Neural network acceleration method in the two-component CPU-GPU computer systems», Proceedings of the 6th International Conference “High

Performance Computing HPC-UA 2020" [on review]. Also, the results of this study were presented at international conferences CSNT-2019, ICSFTI-2019 and published in the relevant proceedings of these conferences.

3. «Zastosuvannia tekhnolohii WCF dlia pidvyshchennia efektyvnosti obchyslen v suchasnykh rozpodilenykh komp'iuternykh systemakh» [The application of WCF technology to increase the efficiency of computing in modern distributed computer systems], Proceedings of the XIII International Conference CSNT-2020 [in press].
4. «Zastosuvannia tekhnolohii WCF dlia pidvyshchennia efektyvnosti paralelnykh obchyslen v khmarnykh rozpodilenykh komp'iuetrnykh systemakh» [The application of WCF technology to increase the efficiency of parallel computing in cloud distributed computer systems], Proceedings of the The International Conference on Security, Fault Tolerance, Intelligence (ICSFTI2020) [in press].

Keywords: parallel computing, distributed computer systems, heterogeneous computer systems, cluster systems, accelerators, calculations scheduling, load balancing, deferred evaluation, deferred evaluation, threads, cores, graphical processing units.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

РКС – розподілена комп’ютерна система.

ПКС – паралельна комп’ютерна система.

РГКС – розподілена гетерогенна комп’ютерна система.

СП – спільна пам’ять.

ЛП – локальна пам’ять.

CPU – центральний процесор (англ. Central Processing Unit).

GPU – графічний процесор (англ. Graphics Processing Unit).

АСТ – абстрактне синтаксичне дерево.

ВСТУП

Позитивну динаміку розвитку інформатики та обчислювальної техніки в тому числі можна спостерігати на всьому проміжку існування цієї галузі. Апаратний та програмний розвиток нерозривно пов’язані, оскільки більш продвинуті алгоритми та програмні комплекси вимагають більш потужної апаратної підтримки. На перших порах основним підходом до збільшення апаратної потужності було удосконалення частотних та електричних характеристик основних складових компонентів комп’ютерних логічних елементів, а також нарощування кількості цих самих елементів на одиницю простору. Однак з часом цей шлях було майже вичерпано [1], оскільки було досягнуто майже граничних характеристик на фізико-хімічному рівні для кремнієвих елементів мікросхем. У зв’язку з цим вектор подальшого розвитку продуктивності обчислювальних систем змістився в бік одночасного застосування спершу одночасно декількох ядер чи процесорів в рамках однієї паралельної системи, а потім і декількох таких систем в рамках однієї розподіленої системи [2].

В той же час, не зважаючи на потенційно безкінечні можливості приросту швидкодії програм в паралельних та розподілених системах, на практиці вони обмежені з математичної точки зору, оскільки існує величезна кількість алгоритмів, які не можуть мати ідеальної паралельної реалізації. Це приводить до того, що коли ці реалізації виконуються на реальних паралельних та розподілених системах, відносна ефективність складових обчислювальних елементів цих систем не максимальна, тому що в роботі кожного з них присутні з тих чи інших причин паузи та затримки. Тому, при детальному аналізі вищеперерахованих факторів, можна прийти до висновку, що при вирішені сучасних наукоємних задач чи організації розподілених серверів масового обслуговування на базі паралельних та розподілених комп’ютерних систем вплив виказує не лише потужне апаратне забезпечення, а й програмні

комплекси, які відповідають за функціонування систем на різних рівнях, від операційного та комунікативного до прикладного [3].

Сучасні комп’ютерні системи високого рівня, в тому числі загальнодоступні користувацькі та мобільні платформи на даний момент будуються за принципами паралельних обчислювальних систем. Програмне забезпечення системного рівня (операційні системи) також будується на основі принципів поділу часу та підтримки багатоядерності та/або багатопроцесорності. Більше того, можна сказати що максимально ефективне вирішення задачі організації паралельної роботи апаратно-програмного рівня є однією з ключових вимог до сучасних операційних систем. Проте, у зв’язку з поясненими вище причинами, навіть паралельності недостатньо сучасним комп’ютерним системам для ефективної роботи. Тому найбільш об’ємі обрахунки, і особливо ті, до яких висовуються ще крім того критичні умови тривалості, виконуються лише на розподілених системах, серед яких найпопулярнішими є кластерні системи [4]. Але останнім часом також значної популярності набувають гетерогенні системи, тобто такі, які містять різнорідні обчислювальні елементи або спеціалізовані розширювальні плати, так звані акселератори обчислень, які являють собою великий набір доступних процесорів простої архітектури. Найпоширенішими із таких плат є графічні процесори, роль яких спершу була в прискоренні центрального процесора шляхом розвантаження його від об’ємних задач роботи з графікою, але з появою шейдерів загального призначення та техніки загального програмування на графічних процесорах (англ. GPGPU - General Purpose Graphics Processing Units) з’явилась можливість використовувати ці плати для будь-яких обчислень. При цьому паралельні системи із акселераторами можуть використовуватись як кінцеві елементи розподілених систем. В такому випадку додатково виникає проблема планування обчислень в таких системах та балансування навантаження в них, з урахуванням неоднорідності, а, відповідно, і різних показників продуктивності вузлів системи.

Наразі існує великий набір стандартних методів балансування навантаження в розподілених системах, але більшість із них орієнтовані на конкретний тип задач, лише невелика кількість не прив'язані до конкретних задач. І тільки на основі декількох із них розроблено адаптації, які б враховували гетерогенність кінцевих вузлів системи. При цьому вони не завжди підтримують більшість технік паралельного програмування, особливо нерегулярний паралелізм. Також характерною рисою їх роботи є необхідність жертвувати або часом, або оптимальністю отриманого розподілу.

Перспективи застосування акселераторів також необхідно дослідити з точки зору використання їх для різноманітних обчислень. Протягом останніх років з'являється все більше наукових статей та дипломних праць присвячених обчисленням в ГКС з акселераторами типу графічних процесорів [5-12], абсолютна більшість із них розглядають дане питання лише з точки зору організації обчислень сугубо на графічних процесорах або ж з точки зору кращого вибору між центральним та графічним процесорами. Особливостям ГКС, в яких одночасно обчислення ведуться і на CPU і на GPU наразі присвячена порівняно невелика кількість робіт. Причому в більшості із них проблематика розглядається в контексті вирішення класичних задач лінійної алгебри, криптографії, та реалізації гіперпаралельних тестових алгоритмів [6-9]. Особливості розгортання нейронних мереж в ГКС CPU/GPU розглянуті лише на прикладі деяких типових задач для нейронних мереж [10], серед яких відсутня одна з найпопулярніших задач – розпізнавання та класифікація образів. Вирішення цієї задачі в рамках гетерогенної комп'ютерної системи представлене в роботі [11], але лише для мобільних систем, які, очевидно, є значно менш розвинутими та продуктивними за класичні стаціонарні системи. Ще в одній роботі [12] розглядалось власне питання пошуку оптимального розподілу навантаження між CPU та GPU, але лише в контексті вирішення типових математичних задач та реалізації тестових паралельних алгоритмів.

Результати попередньо проведених досліджень [13-14], незважаючи на застосування в них максимально спрощених підходів як до вирішення задачі

(побудови нейронної мережі розпізнавання тексту), так і до організації паралелізму, дозволили попередньо підтвердити вірність деяких висунутих гіпотез. В даній роботі також проводиться розвиток цього дослідження, шляхом поширення цих гіпотез на розподілені комп'ютерні системи.

Мета роботи: підвищення ефективності паралельних обчислень в гетерогенних розподілених комп'ютерних системах шляхом розробки методики оптимального розподілу обчислень в ній та балансування навантаження на її елементи.

Об'єкт дослідження: паралельні та розподілені обчислення, високопродуктивні обчислення.

Предмет дослідження: методи організації паралельних обчислень в розподілених гетерогенних комп'ютерних системах, методи планування обчислень та балансування навантаження в РГКС.

Методи дослідження: методи статистичного опрацювання даних, теорія паралельних та розподілених обчислень, теорія планування, теорія оптимізації, теорія компіляторів, теорія алгоритмів, теорія графів.

Ключові слова: паралельні обчислення, розподілені комп'ютерні системи, гетерогенні комп'ютерні системи, кластерні системи, акселератори, планування обчислень, балансування навантаження, відвантажені обчислення, відкладені обчислення, потоки, ядра, графічні процесори.

Автором висловлюється окрема щира вдячність д.т.н., проф. Стіренко С. Г., д.т.н., проф. Новотарському М. А., д.т.н., проф. Сімоненку В. П. та к.т.н., доц. Русановій О. В. за їх цінні поради, зауваження та загальну підтримку роботи.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Загальний огляд підходів до організації обчислень в розподілених гетерогенних комп’ютерних системах

На протязі всієї історії сучасної обчислювальної техніки одними з ключових показників ефективності роботи процесорів та обчислювальних систем на базі них були час, який вони витрачали на пошук рішення тієї чи іншої проблеми, та об’єм пам’яті, виділений для цього. Проте, за останні роки спостерігається значне зростання об’ємів доступної пам’яті в комп’ютерних системах, тому основним критерієм, яким характеризується ефективність роботи таких систем лишився лише час. Відповідно, сучасні алгоритми та підходи оптимізації обчислень націлені саме на мінімізацію цього показнику, як на апаратному, так і на програмному рівнях.

З апаратної точки зору першим підходом до прискорення роботи ядра стало збільшення його тактової частоти. Проте, таке збільшення не безкінечне, і наразі частотні характеристики сучасних чипів сягають максимально можливих показників для кремнієвих чипів. Наступним підходом стало виділення в рамках одного процесору декількох окремих ядер, здатних проводити обчислення одночасно. Системи на базі таких процесорів отримали назву паралельні комп’ютерні системи. Але, окрім значного підвищення вимог до програмного забезпечення, орієнтованого на такі системи, постала також необхідність значного ускладнення інших апаратних компонентів задля забезпечення ефективної роботи кожного наступного доданого ядра. Тому з часом зростання кількості ядер задовольняло потреби пересічних користувачів, але не встигало за потребами науки та бізнесу. Проте в той же час розвиток мережевих технологій не стояв на місці, вони вже були здатні забезпечити швидку передачу досить великих об’ємів даних. Тому наступним вирішенням проблеми прискорення обчислень в комп’ютерних системах стала ідея

розподілення обчислень між декількома окремими однаковими паралельними комп’ютерними системами через певну локальну комунікаційну мережу. Новий клас обчислювальних систем отримав назву кластерні комп’ютерні системи.

Це рішення було певним компромісом. Оскільки з одного боку, витрати для нарощування потужностей в кластерних комп’ютерних системах були значно меншими, ніж витрати для нарощування ядер чи тактової частоти, це дозволяло розгортати величезні системи цього класу, які отримали окрему назву – суперкомп’ютери. Але з іншого боку, ефективне як системне, так і прикладне програмне забезпечення для таких систем ще більше зростало за складністю, порівняно з програмним забезпеченням для звичайних паралельних систем.

Окрім того, з часом організація кластерних комп’ютерних мереж на базі сучасних паралельних обчислювальних систем теж зіткнулась із рядом труднощів. Такі системи займали все більше місця, вимагали потужної системи охолодження та живлення, найпотужніші суперкомп’ютери взагалі розміщувалися в безпосередній близькості від електростанцій. Відповідно організацію та підтримку таких систем могли дозволити собі лише великі наукові центри та бізнес-компанії. Водночас, до того часу розвиток комп’ютерних мереж забезпечив прийнятну швидкість передачі даних через глобальну мережу, що в купі із появою на ринку великого різноманіття дешевих процесорних елементів дало можливість виділити над кластерними системами більш широкий клас систем – розподілені комп’ютерні системи. Елементи таких систем можуть додаватись в необмеженій кількості, при цьому можуть мати різні характеристики, навіть являти собою кластерні системи, і при цьому знаходитися як в одній кімнаті, так і в різних частинах світу.

Ще одним підходом до оптимізації часу виконання програм в комп’ютерних системах стало розподілення обчислень і в середині самої системи. Оскільки процесор являється центральним елементом системи, навіть під час масивного обчислення заданої користувачем задачі він виконує величезну кількість інших, не пов’язаних з поточною задачею обчислень.

Специфіка роботи супервізору операційної системи така, що всі ці завдання, особливо ті, які мають вищий пріоритет, він буде розміщувати на горі стеку завдань. Тобто, користувацьке обчислення, в будь-якому разі, хочемо ми того чи ні, буде постійно ставитися на мікропаузи, а ресурси системи віддаватимуться обробці важливіших завдань. Тому розвантаження процесору шляхом звільнення його від виконання допоміжних та системних завдань шляхом делегування їх до окремих спеціалізованих обчислювачів, названих сопроцесорами або акселераторами, також виказує значну роль в збільшенні швидкості обчислень на центральному процесорі.

Графічний процесор (GPU - англ. graphics processing unit) є одним з найважливіших таких спеціалізованих акселераторів. Його винайдення дозволило звільнити CPU від однієї з «найважчих» його задач – рендерінгу користувацького інтерфейсу. В спрощеному форматі модель GPU можна описати як набір із великої кількості однотипних простих процесорних елементів. Кожен з цих елементів є значно дешевшим від своїх аналогів в CPU. Це сприяло значному розвитку GPU в плані приросту кількості ядер в ньому, і наразі звичайне число ядер в середньостатистичному GPU доходить до півтисячі. Проте, як показала практика, якщо гетерогенна комп’ютерна система, в складі якої є і CPU і GPU, не зайнята роботою зі складною графікою, то така обчислювальна потужність, яка наявна в сучасних GPU є надлишковою, обчислення ведуться так швидко, що більшість часу ядра GPU простоюють в очікуванні наступного завдання від CPU.

Використання цієї вільної обчислювальної потужності, значної та дешевої, є очевидним та логічним вирішенням проблеми збільшення продуктивності обчислень. З винайденням та інтеграцією в GPU програмованих шейдерних блоків з’явилася змога програмування звичайних користувацьких обчислень для GPU. Дана техніка отримала назву GPGPU (англ. General-purpose computing on graphics processing units, неспеціалізовані обчислення на графічних процесорах) і стала важливим проривом в розвитку сучасної комп’ютерної техніки. Перші графічні процесори компанії Nvidia, які підтримували

технологію CUDA (від англ. Compute Unified Device Architecture, уніфікована обчислювальна архітектура пристрою - реалізація техніки GPGPU від цієї компанії) не відрізнялись дешевизною та доступністю. Проте досить швидко з'явились інші реалізації GPGPU, серед яких варто відмітити фреймворк OpenCL. Написаний компанією Apple для мови C++ він розповсюджується під вільною ліцензією та дозволяє програмувати загальні обчислення для GPU не лише від компанії Nvidia, а, наприклад, і компанії AMD. Також, окрім того, що OpenCL підтримує більше GPU ніж CUDA, зазвичай такі GPU дешевші та простіші, аж до звичайних відеокарт.

Отже, винайдення техніки GPGPU та таких технологій як CUDA та OpenCL окрім значного підвищення швидкості обчислень також дало змогу малому та середньому бізнесу, і більшості звичайних інститутів та університетів організувати власні потужні обчислювальні центри, оскільки розподілені комп'ютерні системи, в основі яких лежать гетерогенні системи із CPU та GPU забезпечують достатню швидкість обчислень, і водночас коштують значно дешевше за обчислювальні системи, побудовані лише на великій кількості CPU або GPU. Також, як показує практика, техніка GPGPU стала однією з засад стрімкого розвитку систем штучного інтелекту.

Тому, можна стверджувати, що станом на сьогодні основні засади прискорення вирішення об'ємних задач в комп'ютерних системах зосереджені саме в площині розподілених комп'ютерних систем, в тому числі їх гетерогенних варіантах. А для забезпечення максимально продуктивної роботи таких систем головним фактором, якого необхідно досягти в процесі розробки цільового програмного забезпечення, є збалансований розподіл навантаження на кожну складову паралельну підсистему розподіленої системи, а також на кожен кінцевий процесор або акселератор (за наявності), з врахуванням відмінностей в їх продуктивності.

У випадку незбалансованості показник максимального прискорення, якого можна досягти в таких системах різко падає. Навіть при припущенні однакової продуктивності всіх складових елементів, якщо програма розподілена на P

ядер, то все рівно на протязі деякого часу T_M програма займатиме не більше ніж P_M ядер (P_M<P). Виходячи із закону Амдала [17], максимальне значення коефіцієнту прискорення K_S, яке можна отримати в такій системі становитиме:

$$K_S = \frac{T_1}{T_P} = \frac{T_1}{\frac{T_1 K_P}{P} + \frac{T_1(1-K_P)}{P_M}} = \frac{1}{\frac{K_P}{P} + \frac{1-K_P}{P_M}}\ , \quad (1)$$

Де K_P – коефіцієнт, який показує яка частина обчислень виконується на всіх P ядрах; T_1 – час виконання програми на одному ядрі; T_P – час виконання програми на всіх P ядрах.

При значному збільшенні кількості ядер P, доступних в розподіленій системі, ліміт виведеного виразу відповідатиме максимально можливому коефіцієнту прискорення, який може бути досягнутий в системі [17]:

$$S_{MAX} = \frac{P_M}{1-P}\ , \quad (2)$$

де S_{MAX} – максимально можливий в системі із P ядер коефіцієнт прискорення.

Тобто, максимальне можливе прискорення, яке може бути досягнуто в системі, обмежується зверху незбалансованістю навантаження. Тому саме зменшеність незбалансованості навантаження представляє найбільший інтерес при оптимізації обчислень в розподілених комп’ютерних системах.

1.2. Процес обробки завдань в розподіленій комп'ютерній системі

Для того щоб виділити основні частини розподілених комп'ютерних систем, недоліки в проектуванні та роботі яких впливають на незбалансоване використання доступних обчислювальних ресурсів, слід спершу розглянути весь процес обробки завдань в таких системах.

Перш ніж бути поставленим на виконання конкретним обчислювальним елементом системи, завдання, яке надходить в неї, проходить декілька рівнів оптимізацій засобами різних планувальників, кожен із яких розглядає РКС зі своїх власних рівнів абстракцій. Серед цих рівнів можна виділити наступні:

1. **Рівень планувальника РКС.** На цьому рівні система розглядається як набір неоднорідних складових із абстрагуванням їх до рівня підсистем (кластерних або паралельних). На основі карти системи та певних показників продуктивності складових частин РКС та каналів зв'язків між ними, плануваньник будує загальний граф паралельних задач та проводить оптимальний розподіл задач або підзадач, які були виділені в графі, надсилаючи їх та необхідні їм дані до підсистем. В самих обчисленнях ролі не бере, але може перевідправити задачу на виконання, якщо вона не була оброблена до закінчення певного терміну.
2. **Рівень планувальника кластерної підсистеми.** Складовим елементом РКС може бути кластерна система, тому для таких систем виділяється окремий планувальник. Він розглядає систему як строго однорідну, і виконує рівномірний розподіл отриманої задачі по складовим елементам. В самих обчисленнях ролі не бере.
3. **Рівень планувальника операційної системи.** Прихований від користувача, відповідає за безпосереднє виконання задач на конкретних обчислювальних ядрах, а також за відправку окремих задач до акселераторів. Забезпечує міграції або прив’язки задач до ядер, оптимізацію їх виконання, тощо. Але не може забезпечити міграцію задач за межі машини.
4. **Рівень паралелізму на ядрах.** Деякі сучасні процесори забезпечують виконання декількох потоків на одному фізичному ядрі. Ця технологія отримала назву Hyperthreading. Переключення потоків в межах ядра відбувається на апаратному рівні, і є більш прозорим для операційної системи, відбувається більш швидко, ніж програмна міграція потоків між ядрами, а також може відбуватися за апаратними подіями (наприклад промахами кешу).
5. **Рівень паралелізму команд.** Більшість сучасних процесорів, які використовуються в РКС, є суперскалярними. Кожне ядро такого процесору здатне виконувати декілька команд за один системний такт.

Згідно з професором Стіренко, весь процес обробки задач в РГКС можна представити схемою, зображеною на рисунку 1.1 [17].

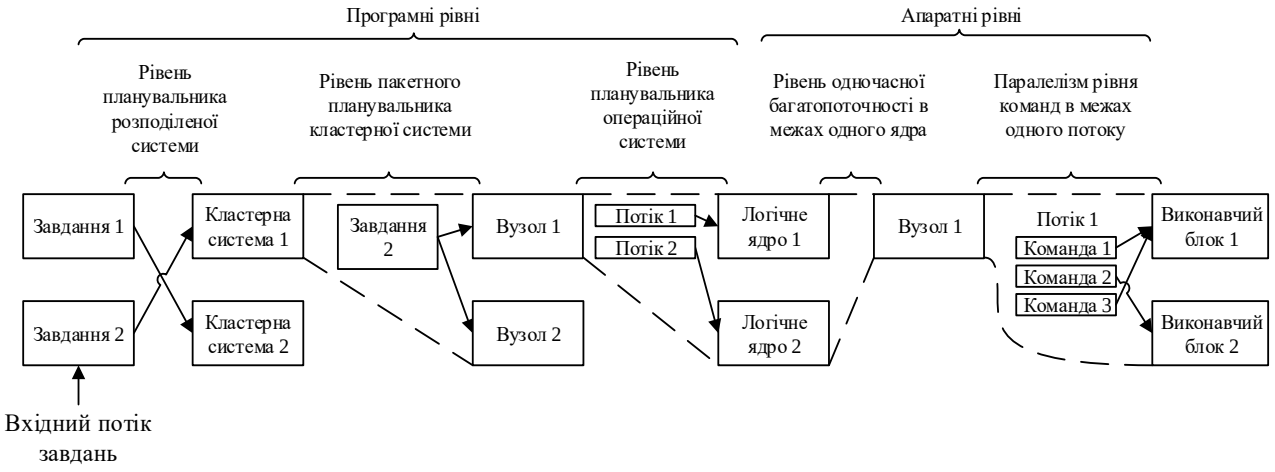

Рис. 1.1. Схема процесу обробки завдань в розподіленій паралельній (кластерній) обчислювальній системі

З наведеного вище аналізу можна зробити висновок, що при такій моделі обробки задач в РКС ресурси, які необхідні для цього, виділяються один раз перед запуском і звільнюються всі разом після завершення її виконання. Задача не може звільнити ті ядра, які стали не потрібні через завершення її виконання, або ж надати запит на виділення їй більших обсягів ресурсів. Окрім того, не враховується варіант, коли програмістом явно задано розділення певних задач на підзадачі в процесі виконання (як наприклад в Fork-Join моделі), що також тягне за собою незбалансованість навантаження, яка виявиться лише в процесі виконання, тому буде пропущена планувальниками розподіленої, кластерної та паралельної систем. Тому і виникає задача балансування навантаження в РКС.

Умовно балансування навантаження можна розділити на статичне та динамічне. Статичне балансування відбувається перед безпосереднім запуском задач на виконання, та базується на структурі задач, а також алгоритмах оцінки, планування та передбачення навантаження та продуктивності. Динамічне ж балансування навпаки, відбувається вже в процесі роботи задачі, та не використовує передбачення часу виконання задачі. Воно має більш широку сферу застосування через можливість роботи з задачами з нерегулярним паралелізмом. В цілому, таке балансування можливе і автоматично, на рівні

планувальника операційної системи, і стає ще більш ефективним, якщо перейти в програмі від концепції потоків до концепції задач, цим самим водночас зменшивши зерно паралелізму. Задачі, які є більш легковісними, ніж повноцінні потоки, можуть операційною системою плануватись та переміщуватись більш швидко, ніж потоки. Проте, не для всіх задач є доцільним перехід від потокової моделі паралелізму до моделі, заснованої на задачах. Окрім того, менеджмент задач все рівно може відбуватися лише на рівні планувальника операційної системи, що унеможливлює перерозподіл виконання на більш високих рівнях в процесі роботи.

Отже, необхідно забезпечити можливість автоматичного оптимального завантаження елементів та автоматичного розподілу задач на більш високих рінях в системі, не лише перед, а й вже безпосередньо в процесі виконання. Жоден із виділених на рис. 1.1 рівнів обробки задач в РКС не включає в себе динамічного балансування навантаження. Тому необхідно спершу виділити рівні, на яких можливо та доцільно застосовувати таке балансування, а далі визначитись із тим, що являтиме собою одиницю балансування навантаження в системі.

Як статичне, так і динамічне балансування навантаження не можливі без попереднього аналізу задачі та оцінок різних часових показників в системі. При цьому на результати цих аналізів та показників в значній мірі на них впливатиме те, яка модель паралельного програмування застосовується та підтримується в конкретній РКС.

1.3. Огляд сучасних підходів до паралельного програмування

Однією з основних сучасних тенденцій в розробці і прикладного і навіть системного програмного забезпечення є намагання перейти до оперування максимально можливими рівнями абстракції. В значній мірі цьому сприяє також сучасний розвиток та популярність високорівневих мов програмування, які дозволяють розробникам прикладного програмного забезпечення

зосереджуватись на розробці логіки програми на таких рівнях синтаксису і семантики, які більш наближені до предметної області, а не до апаратних та системних складових. Основний недолік такого підходу полягає в тому, що це позбавляє розробника можливостей взаємодії з системою на найнижчих рівнях, аж до апаратного, і таким чином проводити тонку оптимізацію програми. Проте переваги компенсують цей недолік. Серед них підвищення швидкості розробки програмного забезпечення, розширення можливостей до його повторного використання, портабельності та сумісності, поява можливостей для автоматичного тестування, відлагодження та оптимізації, оскільки запис програми в більшій мірі описує що вона має робити, а не як саме вона має це робити.

В паралельному програмному забезпеченні ця тенденція також знаходить відгук, в сучасних мовах та бібліотеках паралельного програмування з’являється все більше високорівневих абстракцій, які, наприклад, дозволяють програмісту відійти від концепції важких повномірних потоків до більш легких задач, або ж навіть лише виділити потенційно можливі та бажані до паралельної обробки ділянки послідовної програми (в найпоширенішому випадку - ітерації циклів), а компілятор сам організує відповідний паралелізм. Окрім того, незважаючи на те, що автоматичний паралелізм є наразі надзвичайно складною задачею, але сучасні оптимізуючі компілятори містять певні його задатки, і навіть без команд від розробника можуть забезпечити автоматичну паралельну обробку деяких бажаних до розпаралелювання ділянок програм.

Проте, такі підходи, хоч і розвиваються з року в рік, але все ще можливі лише в паралельних системах зі спільною пам’яттю, які, як було вказано вище, мають наразі значне обмеження на додавання нових процесорних елементів, і тому на сьогоднішній день використовуються лише як кінцеві вузли розподілених паралельних систем із локальною пам’яттю. Та, не зважаючи на це, такі підходи демонструють вкрай високу ефективність, що робить актуальною задачу адаптації їх до використання в системах з локальною

пам’яттю. І, оскільки такі системи зазвичай розподілені, незважаючи на самі високі рівні абстракції, з програмної точки зору вони являють собою набір окремих процесів. Тому постає питання організації ефективної взаємодії цих процесів під час обробки задачі.

1.4. Огляд сучасних моделей взаємодії паралельних процесів

Існують дві основні моделі взаємодії паралельних процесів, виконання яких для повноти функціонування має забезпечувати будь-яка паралельна система. Це:

- Передача даних;
- Синхронізація паралельних процесів (потоків, задач, тощо).

1.5. Огляд сучасних моделей паралельного програмування

На програмному рівні можна виділити дві базові моделі взаємодії паралельних процесів: модель, заснована на спільних змінних, та модель, заснована на передачі повідомлень. Кожна з них втілює обидві основні операції та забезпечує повноцінне функціонування паралельної системи. На практиці ці моделі є основою, в концепції якої відбувається реалізація остаточної моделі програмування, прийнятої в системі. Розглянемо їх детальніше.

1.5.1. Модель з статичним створенням потоків

Така модель в першу чергу орієнтована на реалізацію в паралельній комп’ютерній системі зі спільною пам’яттю. Полягає в ручному виділенні програмістом в рамках програми декількох потоків, і наданні йому стандартизованого інтерфейсу їх менеджменту, серед основних функцій якого можна виділити наступні:

- Створення потоку

- Запуск потоку на виконання
- Тимчасове блокування потоку
- Поновлення роботи потоку
- Примусове завершення роботи потоку
- Перевірка поточного стану потоку
- Очікування завершення роботи потоку

На основі деяких з цих операцій надаються також більш абстрактні ті чи інші програмні засоби для організації взаємодії потоків, в тому числі для вирішення задачі взаємного виключення та синхронізації. Серед них основними являються семафори, м'ютекси, синхронні методи, монітори, замки, бар'єри, події та атомарні операції і змінні. В паралельній програмі можуть виникнути такі небажані ситуації, як тупики, гонки та відсутність гарантії завершення.

Проблема виникнення тупиків

Ситуації, при яких один або декілька потоків нескінченно довго очікують на певний сигнал від іншого чи інших потоків, і, відповідно, ніколи не зможуть завершити свою роботу, називаються тупиковими. При ручному створенні потоків, і, відповідно, ручному розміщенні в програмному коді елементів їх взаємодії, людський фактор є головним чинником виникнення подібних ситуацій. Випадки, коли розробник не в повній мірі передбачив всі можливі варіанти виконання потоків, або неправильно розставив операції захоплення та звільнення потокам елементів їх синхронізації і приводять до виникнення тупиків в роботі паралельних програм. В цьому випадку потоки призупиняють свою роботу, цим самим звільнюючи обчислювальні потужності машини, і поновлені будуть лише операційною системою. Проте можлива і ще гірша ситуація – виникнення динамічного тупику.

Проблема виникнення динамічних тупиків

Особливим випадком тупика є динамічний тупик, при якому потоки не блокуються, а продовжують свою роботу, безкінечно один за одним взаємно змінюючи свій стан, і, відповідно, при цьому не маючи змоги перейти до подальшої роботи. Також динамічні тупики можуть виникати при залежності

роботи потоків від часових функцій або навіть в банальних випадках загальних програмістських помилок, як то організація безкінечних циклів чи неправильних переходів.

Проблема голодування потоків

Іншою поширеною проблемою, яка може виникнути при ручному створенні потоків є їх так зване «голодування». Це ситуація, при якій деякі потоки в паралельній програмі необмежено довго очікують доступу до певних спільних ресурсів, доступ до яких за логікою програми для цих потоків передбачено, але не надається через неправильну його організацію.

Проблема відсутності гарантій виконання

Ця проблема є наслідком двох попередніх. Сама по собі гарантія виконання (або ж гарантія прогресу) це властивість паралельної програми досягти свого завершення не залежно від того, яким чином було сплановано потоки та їх синхронізацію. Відповідно, при ручному створенні потоків та внесенні для їх синхронізації в програму блокуючих елементів синхронізації, гарантія виконання відсутня як така, тобто можливе виникнення ситуацій, коли через неможливість завершення одного або декількох потоків програма сама по собі не зможе завершитись. Операційна система безкінечно довго може видавати кванти часу потоку, який перебуває в заблокованому стані, і тим самим тримає блокування всієї програми.

Модель з ручним створенням потоків є першою винайденою та найбільш застосовуваною моделлю. Через це ведуться постійні розробки в сфері передбачення, уникнення та усунення описаних вище проблем в паралельних програмах, розроблених за даною моделлю. Проте все ще можливість виникнення помилок в таких програмах лишається дуже високою.

Також значним недоліком цієї моделі є необхідність частого застосування блокуючих засобів синхронізації, через що в паралельній програмі виникають послідовні ділянки виконання, що суперечить самій ідеї розпаралелювання та, згідно з розглянутим вище законом Амдала, вносить принципове верхнє

обмеження максимально можливого для досягнення коефіцієнту прискорення таких програм.

Головною перевагою цієї моделі є надання можливості програмісту працювати з низькорівневими примітивами, що з одного боку дозволяє найбільше наблизитись саме до особливостей цільової системи, а з іншого, наприклад, дозволяє вручну організувати і модифікувати будь-яку із описаних далі моделей.

1.5.2. Модель з динамічним створенням потоків

Модель з динамічним створенням потоків є однією з найбільш складних та, водночас, найбільш потужних моделей розпаралелювання. Перш за все це пов’язано з тим, що нема ніяких обмежень на способи організації паралелізму, організації обчислень, архітектуру програми в цілому.

Програмісту надається лише обмежений найбільш поширеними засобами набір варіантів створення потоків, станів створених потоків, засобів взаємодії. Здебільшого вимагається лише описати шаблон одного робітничого потоку та динаміку створення та знищення таких потоків. Відповідно, при цій моделі вже завжди присутня гарантія прогресу та уникається значна кількість можливих тупикових ситуацій.

Найпоширенішою такою моделлю є рекурсивна модель Fork-Join. Вона базується на двох операціях: Fork – розділення батьківського потоку на декілька (зазвичай, але не обов'язково, два) дочірніх потоків, виконання операцій в цих потоках (в тому числі це також можуть бути операції Fork-Join) та Join - повернення і об’єднання результатів роботи дочірніх потоків у батьківському потоці.

Дана модель в тому чи іншому вигляді реалізована у великій кількості сучасних мов та бібліотек паралельного програмування, серед яких Java (пакет concurrent), C/C++ (бібліотеки OpenMP, Futures, надбудована мова Cilk), Golang

(goroutines), C# (деякі засоби бібліотеки TPL, як то задачі (Tasks), паралельні цикли, паралельні ітератори).

Ця модель також не позбавлена недоліків. Одним із них виступає інколи занадто сильна обмеженість програміста в засобах, що призводить до неможливості ефективно розв'язувати задачі певного класу із застосуванням цієї моделі. Наприклад, в бібліотеці OpenMP версії 2.0 важко розв'язувати задачі із нерегулярним паралелізмом. Регулярний паралелізм в ній повноцінно підтримується директивою pragma omp for, проте для нерегулярного паралелізму доступна лише директива pragma omp section, яка дозволяє виконувати лише фіксований набір команд і лише у фіксованій кількості потоків, що, відповідно, не дозволяє в процесі виконання змінювати зерно паралелізму.

Для вирішення цієї проблеми в 2011 році в мову C++ було додано бібліотеку Futures [26], яка дозволяє одним потокам делегувати роботу іншим. Проте в цій бібліотеці все ще відсутні можливості динамічної регуляції зерна паралелізму, тобто будь-які можливості організації такого механізму можуть бути закладені розробником лише на етапі проектування, в залежності від цільової задачі.

Засоби мови розширення Cilk та бібліотеки OpenMP 3.0 (Tasks) майже ідентичні засобами бібліотеки Futures, тобто в цілому зберігають вище як переваги, так і недоліки.

1.5.3. Модель з автоматичним створенням потоків

Хоч, як було показано вище, обмеженість способів створення, взаємодії та завершення роботи потоків, є не тільки перевагою, а й, в деяких випадках, недоліком, все ж ця ідея знаходить і подальший розвиток, який є зручним для великого класу задач. Полягає вона в повному обмеженні можливостей будь-якого ручного створення потоків, як в статичному, та і в динамічному режимі. Майже вся відповідальність за організацію і підтримку паралельної роботи

програми покладається на допоміжні бібліотеки часу виконання та оптимізуючі і розпаралелюючі компілятори.

Роль оптимізуючих і розпаралелюючих компіляторів полягає в автоматичному додатковому аналізі коду для знаходження потенційно паралельних ділянок та організації їх коректного паралельного виконання. Зазвичай більшість сучасних компіляторів високорівневих мов програмування виконують цю функцію автоматично. Недоліків тут можна відмітити декілька:

1. Цей підхід неможливо застосувати до інтерпретуємих мов програмування. Навіть за додатковим введенням оптимізатора нівелюється одна з основних переваг застосування інтерпретатора замість компілятора – відсутність необхідності виділення часу та засобів на компіляцію. Також інтерпретатор має на увазі покрокове виконання програми, тобто потенційно він не має жодної інформації про наступні кроки, особливо про роботу з даними, а аналіз синтаксичних дерев не лише за операціями, а й за даними є ключовим для організації коректного паралелізму;
2. Задача аналізу синтаксичних дерев за даними є NP-повною, тобто можливо знайти лише наближене її рішення. Відповідно, в програмах, в яких паралелізм введено засобами автоматичного розпаралелювання компілятором можливі ситуації, коли кінцевий результат недостатньо розпаралеленим або містить занадто багато взаємодій (і, відповідно, критичних секцій), що в обох випадках веде до зменшення швидкодії.

Виправити недоліки, описані вище, покликаний підхід до автоматичного розпаралелювання із ручним вказанням можливих паралельних ділянок в програмі. Відповідно, бібліотекою часу виконання або компілятором чи оптимізатором ведеться дослідження перш за все саме всередині цих блоків, що вже значно зменшує затрати та збільшує вірогідність отримати максимально ефективне рішення. Прикладом такого блоку можна навести операцію Parallel.Invoke із стандартної бібліотеки TPL мови C#.

В обох випадках у разі виявлення можливостей до розпаралелювання програма сама вирішує яку кількість потоків створити та як організувати взаємодію між ними.

Переваги такого підходу очевидні:

1. Процес розробки паралельних програм значно спрощується та не вимагає від розробників значної спеціальної кваліфікації;
2. Можливість автоматичної зміни зерна паралелізму в програмі.
3. Спрощені передумови до такої зміни.

Основним недоліком цього підходу, як вже було підмічено вище, є складність роботи з даними. Як таке, поняття даних взагалі відсутнє в цій моделі, тому потенційно можливі випадки некоректної роботи з ними, або ж відсутності можливості автоматичного паралелізму з такими даними (чого можна було б уникнути, якби розбиття даних проводилось вручну, відповдно до однієї із описаних вище моделей).

1.5.4. Модель з транзакційною пам’яттю

З описів попередніх моделей стає зрозуміло, що питання організації коректної взаємодії із даними є ключовим для ефективної орагіназції паралельних програм. І проблемою є не лише уникнення власне конфліктів доступу до даних, а й некоректних результатів, особлива загроза виникнення яких можлива при автоматичному створюванні потоків. Одним із вирішень описаної проблеми є застосування транзакційної пам’яті, яка являє собою розширення концепції спільних змінних. Оскільки зчитування даних є потокобезпечною ситуацією, то будь-який потік потім може зчитати дані без жодних обмежень та провести з ними певні маніпуляції чи обрахунки (почати транзакцію). Проте якщо результат транзакції впливає на ці дані, то він записується лише тоді, якщо інші потоки в цей час не надали результати своїх транзакцій з цими даними, тільки тоді транзакція вважається завершеною. У разі ж, якщо декілька потоків бажають записати результати своїх транзакцій, то

всі ці тразакції відхиляються та операції виконуються повторно, допоки не буде достягнуто безконфліктного порядку записів.

Переваги цієї моделі наступні:

1. Можливість вирішення задачі взаємного виключення без необхідності блокування потоків;
2. Можливість загального спрощення вирішення задачі взаємного виключення, оскільки не треба вручну створювати критичну ділянку;
3. Застосування цієї моделі унеможливлює виникнення тупиків.

Серед недоліків можна виділити потенційну можливість тривалого відхилення транзакцій у разі активної роботи багатьох потоків із невеликою кількість одних і тих же даних. Також варто зазначити, що, хоч реалізація транзакційної пам’яті можлива і лише на програмному рівні, проте ефективне її функціонування можливе лише за наявності відповідної апаратної підтримки. Така реалізація присутня наприклад в кластерах IBM BlueGene/Q. Серед користувацьких процесорів транзакційна пам'ять присутня в деяких процесорах Intel від 2014 року випуску.

1.5.5. Модель низькорівневої передачі повідомлень

Розглянуті вище конфлікти з даними виникають через те, що декілька потоків намагаються вести запис в одну і ту ж адресу пам’яті. Це пов’язано з тим, що, хоч кожен потік має власний стек, проте йому не виділяється власний адресний простір. Спершу це було пов’язано в цілому із значними обмеженнями та дороговизною пам’яті в попередніх поколіннях комп’ютерних систем. Проте наразі, із значним здешевінням пам’яті, знаходяться можливості до виділення власного адресного простору кожному з потоків. Оскільки кожен потік веде операції сугубо в своєму адресному просторі, то відпадає необхідність введення критичних ділянок в код.

Проте, максимального прискорення відповідно до закону Амдала досягти все ще не можливо. Тому що виділення власних адресних просторів породжує і

проблему, оскільки необхідно переносити або копіювати дані із адресного простору одного потоку до адресного простору іншого потоку. І хоч потоки не блокуються під час таких операцій, але обчислення не відбуваються допоки дані не будуть перенесені. Тобто, час роботи такої програми прямопропорційний до об’єму даних і частоти їх переносів.

Операції обміну даними традиційно називаються повідомленнями (від одного потоку до іншого). Для повноти функціонування моделі достатньо двох таких операцій – прийому повідомлення та відправлення повідомлення. Модель створення потоків в цьому випадку в більшості випадків статична, рідше динамічна, і майже ніколи – автоматична.

Традиційно ця модель знаходить застосування у системах із локальною пам’яттю та розподілених системах (тому що по суті є основоположною та природньою для таких систем). Проте немає ніяких перешкод до реалізації цієї моделі і в системах зі спільною пам’яттю, питання полягає лише в тому, чи буде це ефективніше за модель, засновану на єдиному адресному просторі та критичних ділянках.

Незаперечною перевагою даної моделі в системах з локальною пам’яттю є її низькорівневість. Програмісту надається максимум засобів організації та контролю за комунікаціями та виконанням програми, на основі яких вже можна будувати більш високорівневі моделі для систем з локальною пам’яттю. Програма в цьому випадку за своєю структурою і логікою максимально наближена саме до системи.

Проте в цьому полягає і основний недолік – програма відображає перш за все структуру системи, а не проблематику та логіку цільового завдання. До того ж, програмний код з низькорівневими примітивами зазвичай більш об’ємний. Також варто відмітити те, що застосування деяких низькорівневих примітивів може призводити до вже описаних вище тупиків, голодувань потоків та відсутності гарантій прогресу.

1.5.6. Модель програмування без блокувань

Наявність блокувань потоків при обробці критичних секцій в моделях, заснованих на спільних змінних, веде до ряду недоліків:

1. Обмеження розширюваності системи. Додавання кожного нового процесору та/чи потоків веде до появи все нових критичних ділянок, тим самим збільшуючи сумарний час послідовної роботи програми, що за законом Амдала веде до зменшення максимально можливого коефіцієнту прискорення, що можна досягти в такій системі. Тобто, починаючи з певного моменту ефективність кожного нового доданого паралельного елементу буде меншою від усіх попередніх;
2. Створення нових можливостей до виникнення тупиків. Розширення системи також створює можливості до появи нових тупикових ситуацій, особливо при моделі зі статичним створенням потоків;
3. Проблема вибору співвідношення об’єму захищених даних та кількості елементів синхронізації. У разі великої кількості спільних даних доцільно розділити їх на декілька різних критичних секцій, аби зменшити сумарний час роботи програми в послідовному режимі. Але при цьому необхідно прямопропорційно збільшити кількість елементів синхронізації в програмі, що веде до нераціонального використання системних ресурсів. При цьому розв’язання цієї проблеми покладено цілком і повністю на розробника.

Альтернативним підходом є повна відмова від блокувань в програмі, досягти якої можливо шляхом використання алгоритмів вирішення задач синхронізації та взаємного виключення, які не вимагають використання блокування. Відповідно до сили гарантій прогресу та наявності очікування потоків виділяється [18] три основні групи неблокуючих паралельних алгоритмів:

1. Алгоритми без перешкод (англ. obstruction-free). Надає найбільш слабкі гарантії, які обмежуються лише умовою, що кожен потік гарантовано завершиться за певну скінченну кількість кроків, при цьому немає

обмежень на максимальну кількість цих кроків. Не надається жодних гарантій прогресу у випадку, якщо потоки працюють зі спільними даними;

2. Алгоритми без блокування (англ. lock-free) надають гарантії прогресу обчислень в системі в цілому, оскільки виключається сама можливість появи тупикових ситуацій. При цьому не надається гарантій відсутності голодування потоків;
3. Алгоритми без очікування (англ. wait-free) надають найсильніші гарантії прогресу, оскільки кожна окрема операція в таких алгоритмах закінчується не пізніше певного заданого кванту часу. При цьому квант часу не має максимального обмеження. Тому це лише частково дозволяє гарантувати відсутність тривалого голодування потоків.

За аналогією з використанням семафорів, м’ютексів, моніторів та інших засобів синхронізації в блокуючих паралельних алгоритмах, для реалізації неблокуючих паралельних алгоритмів використовуються атомарні змінні та операції. З точки зору потоків та операції виконуються як єдине ціле, або ж не виконуються взагалі. Неподільність цих операцій гарантується на апаратному рівні.

Реалізація та семантика атомарних операцій строго прив’язана до моделі пам’яті, яка застосовується в обраних для конкретної паралельної комп’ютерної системи засобах організації паралельних обчислень. Відповідно до цієї моделі визначається, які операції запису видні для операцій зчитування, які проводяться з інших потоків. Тому виходячи з цього можна відмітити головний недолік моделей паралельних обчислень без блокувань – високі вимоги щодо кваліфікації розробників таких алгоритмів.

1.6. Порівняння моделей паралельного програмування

Зі всього сказаного вище слідує, що кожна модель паралельного програмування має як свої переваги, так і недоліки. Зведемо їх в єдину таблицю (таблиця 1.1).

Таблиця 1.1

Порівняльна таблиця моделей паралельного програмування

Характеристика моделі паралельного програмування	Модель паралельного програмування					
	Створення потоків			Транзакційна пам'ять	Низькорівнева передача повідомлень	Без блокувань
	Статичне	Динамічне	Автоматичне			
Підтримка в ПКС з СП	Так	Так	Так	Так	Так	Так
Підтримка в ПКС з ЛП	Ні	Ні	Ні	Ні	Так	Ні
Робота зі спільними ресурсами	Задача взаємного виключення		Прихована від розробника	Апаратні або програмні транзакції	Спільні ресурси відсутні	Неподільні апаратні операції
Статична регуляція зерна паралелізму	Можлива, розмір зерна завчасно декларується розробником					
Динамічна регуляція зерна паралелізму	Ні	Потребує окремого підходу для кожної задачі	Так, розмір зерно регулюється автоматично	Ні	Ні	Ні
Автоматична регуляція зерна паралелізму	Ні	Ні	Так	Ні	Ні	Ні

1.7. Огляд сучасних технологій програмування для паралельних та розподілених систем

Динамічний розвиток паралельних та розподілених систем вимагає створення нових та дослідження існуючих моделей і технологій їх програмування. З проведених вище аналізів випливає, що найважливішим є питання вибору рівню абстракції в прийнятій моделі. Тому що, з одного боку застосування низькорівневих абстракцій потенційно забезпечує максимально ефективну роботу системи, тому що розробник оперує перш за все поняттями, наближеними до системи. При цьому вимагається наближення задачі до абстракцій системного рівня. Це лише теоретично веде до значного прискорення. Через значну пов'язаність системного та прикладного рівнів

принципові помилки можуть виникати на будь-якому з них. З іншого ж боку, застосування високорівневих абстракцій дозволяє оперувати поняттями, наближеними до прикладного рівня та/або бізнес-логіки, що дозволяє звільнити розробника від організації рівня системної взаємодії, поклавши її вирішення на систему. Відповідно і принципові помилки можуть виникати лише на прикладному рівні.

Найсучасніші підходи до організації паралельних обчислювальних систем все більше опираються на високорівневі абстракції, маючи на меті дозволити розробнику оперувати математичними об’єктами та об’єктами бізнес-логіки, використовуючи паралелізм лише на загальному рівні, без вимоги ручної організації всіх комунікацій, синхронізацій, тощо. Деякі сучасні математичні бібліотеки дозволяють виконувати функцію як в послідовному, так і в паралельному режимі, без зміни інтерфейсу цієї функції [19]. Існують підходи які дозволяють описувати паралелізм лише на абстрактному рівні задач та зв’язків між ними [20-21]. Однак, всі ці описані підходи орієнтовані на виконання лише в системах зі спільною пам’яттю [17].

В той же час, як було показано вище, найперспективніші паралельні системи будуються за принципами розподілених систем, при чому системи з акселераторами також за своїм принципом функціонування є розподіленими системами. А з аналізу, приведеного в таблиці 1.1, слідує, що такі системи будуються за моделями, які відповідають системам з локальною пам'яттю.

Адаптація моделей з високим рівнем абстракції в таких системах вимагає від програмного забезпечення, яке реалізує функціонування таких систем, вирішення цілого ряду задач, таких як забезпечення автоматичного розбиття та передачі даних, передачу цих частин конкретним вузлам системи, за необхідності збір результатів від цих вузлів. При цьому необхідно враховувати потенційну гетерогенність системи, тобто можливість наявності в системі складових елементів (в тому числі акселераторів) з різними характеристиками, а, відповідно, і різною потужністю, а також продуктивність комунікативних каналів між елементами системи.

Окремо необхідно виділити задачу доступності даних, яка є ключовою. В системах зі спільною пам’яттю, у зв’язку з використанням спільного адресного простору всіма потоками, ця задача значно спрощується, тому що дані доступні для всіх потоків за замовчуванням. А в системах з локальною пам’яттю та/або акселераторами додатково необхідно вручну забезпечувати доступність даних, шляхом копіювання їх в пам’ять обчислювального елементу системи. При цьому часто недоцільно копіювати всі дані в пам’ять всіх елементів, оскільки ідея розподілених обчислень полягає в виконанні окремих частин програми на окремих вузлах, і часто ці частини використовують лише певну частину даних. Передавати повністю всі дані було б простіше, але зазвичай мережевий час до 10000 разів дорожчий за процесорний, тому чим менші обсяги даних будуть курсувати системою, тим меншими будуть простої обчислювачів. Тому планувальники обчислень намагаються розташувати дані таким чином, щоб забезпечити максимальну локалізацію даних, тим самим максимально зменшити комунікативні затримки системи в цілому. Однак існує широкий клас задач, особливо серед лінійної алгебри та теоретичної фізики, під час виконання яких цільовий елемент для передачі даних, а також об’єм цих даних визначаються в процесі виконання, що перешкоджає роботі планувальника. Тому виникає необхідність передачі йому даних про задачу, джерела передачі даних, обсяги цих даних. Сучасні моделі та технології організації паралельних та розподілених обчислень підтримують цей підхід на тому чи іншому рівні. Тому надалі необхідно розглянути принципи організації паралельних процесів та їх доступу до даних, які застосовуються в найпопулярніших сучасних технологіях як паралельного, так і розподіленого програмування.

1.7.1. Технологія розпаралелювання з використанням директив препроцесору

Як було описано вище, наразі спостерігається тенденція до використання паралельних абстракцій більш високого рівня, які не вимагають значної зміни

інтерфейсу і структури програми, а також високої кваліфікації, дозволяючи зосередитись на вирішенні математичної проблеми. Тому однією з найпопулярніших технологій паралельного програмування для систем зі спільною пам’яттю стала OpenMP. Шляхом невеликих змін в послідовному коді програми ця технологія дозволяє організувати паралельне його виконання. Проте, ефективне її застосування вимагає від розробника ручного позначення залежності між даними та різними частинами програми. За основу цієї технології взято розширення компіляторів мов C та C++, реалізоване у виді бібліотеки, яка наразі вже здебільшого поставляється у складі стандартного набору цих мов. Дана бібліотека розширює набір стандартних директив препроцесору (компілятору), надаючи додаткові директиви для помітки різноманітних паралельних секцій. Також є реалізації цієї бібліотеки для більш сучасних високорівневих мов програмування, таких як C# або Java, але їх ефективність нижча, ніж ефективність вбудованих засобів організації паралелізму в цих мовах.

Оскільки внесення директив до вже існуючих програм не вимагає значних модифікацій вихідних кодів, то ця технологія знаходить широке застосування і при введенні паралелізму до вже існуючих програм. Більшість сучасних трансляторів мають підтримку директив препроцесору, які постачаються бібліотекою OpenMP 3.0 [22-24]. Під час розробки OpenMP було враховано результати попередньо проведених збору та аналізу статистичних даних складних наукових обчислень та реалізацій алгоритмів. Ці результати показали, що регулярний паралелізм та виконання циклічної обробки ітеративних складових алгоритмів займає в середньому до 80% часу виконання програм [25]. Через це в оновленій версії основна увага була зосереджена на реалізації механізмів розпаралелювання циклічних конструкцій.

Як було визначено вище, для ефективної роботи паралелізму важлива не тільки його пряма реалізація, а й підтримка, оптимізація до особливостей архітектури цільової системи та реалізованого алгоритму, і все це в тому числі в динамічному режимі, відповідно до умов, які виникають в процесі роботи.

Для цього в OpenMP було включено бібліотеку часу виконання, на яку було покладено ці обов’язки. Окрім механізмів організації самого паралелізму до цієї бібліотеки було включено також потужні механізми як статичного планування (тобто такого, яке відбувається до старту обчислень), так і динамічного (тобто такого, яке відбувається безпосередньо під час обчислень). В деяких конструкціях бібліотеки розробнику надається можливість вибрати той тип планування, який він вважає доцільним для конкретної ситуації. Варто зазначити, що ці механізм планування не заміщують планувальники операційної системи, а лише доповнюють їх, маючи змогу більш гнучко керувати ресурсами, виділеними системою для програми, при цьому зосереджуючись саме на оптимізації роботи програми, а не системи в цілому.

Також OpenMP визначає ряд спеціальних вказівок, які відповідають за різні режими доступу до даних в паралельних програмах. Розробникам дозволяється позначити дані як:

1. Заборонені до доступу з усіх потоків;
2. Спільні для всіх потоків (копіювання не відбувається);
3. Розподілені між всіма потоками шляхом копіювання, доступні лише на час виконання потоків;
4. Розподілені між всіма потоками шляхом копіювання, оригінал змінної модифікується лише з того потоку, який закінчить роботу останнім;
5. Розподілені між всіма потоками шляхом копіювання, оригінал змінної модифікується за результатами роботи всіх потоків з виконанням редукування;
6. Розподілені між всіма потоками шляхом виділення відповідної пам’яті, але з очікуванням ручного опису особливостей процесу копіювання.

Загалом порядок доступу до даних може бути різним для різних структур даних. Також можна задавати спосіб доступу за замовчуванням. При необхідності система сама може проводити копіювання даних, без явного виклику цієї операції користувачем. Проте для уникнення непередбаченої

поведінки програми використання спільних ресурсів для багатьох потоків все ж вимагає ручного визначення критичних секцій.

Останні дві особливості визначають орієнтацію технології OpenMP на роботу в системах зі спільною пам'яттю. Тому при роботі системи вважається, що швидкість доступу до даних та швидкість їх копіювання однакова для кожного потоку, як однакові і абсолютні показники продуктивності кожного ядра системи. Хоча технологія і може бути застосована в системах із неоднорідною пам'яттю (NUMA-системах), врахування неоднорідності пам’яті не буде відбуватись, що може виказати значний вплив на швидкодію програми в цілому. На цю орієнтацію налаштована і внутрішня підсистема керування ресурсами, яка використовує підхід розподілення робіт шляхом викрадення (англ. work-stealing), який полягає в перетворенні програми в набір готових до виконання підзадач та формуванні черги їх виконання.

Для адаптації технології розпаралелювання на основі директив препроцесору до використання в більш перспективних наразі системах із локальною пам’яттю пропонуються два різних підходи.

Перший з них полягає в автоматичному переході до технології на базі інтерфейсу низькорівневої передачі повідомлень. Досягається це шляхом організації додаткового транслятору, який забезпечить перетворення коду стандарту OpenMP в код стандарту MPI. Окрім принципової складності самих таких трансляторів, це викликає цілий ряд складнощів і самого процесу трансляції, більшість з яких пов'язані з необхідністю копіювання даних в локальну пам'ять вузлів. Оскільки OpenMP не підтримує зміну змісту операцій в процесі виконання, то треба забезпечити найбільш строги рівень несуперечливості – послідовну несуперечливість. Це приводить до падіння ефективності системи в цілому, оскільки веде до великих накладних затрат та організацію міжпотокової взаємодії та очікування потоками даних. При цьому не враховується потенційна гетерогенність системи, оскільки для передачі даних використовуються блокуючі колективні передачі даних MPI, які

займають стільки часу, скільки необхідно для передачі даних між двома комунікативно найвіддаленішими вузлами.

Другий підхід не вимагає перетворень вихідних кодів, а використовує лише розширення власне трансляторів мов C/C++, які реалізовані також в рамках спеціалізованих бібліотек для кластерних систем від компанії Intel. В його основу покладено аналіз самого коду на предмет доступу до даних з різних частин програми, і за результатами цього аналізу формуванні найоптимальніших передач лише тих даних, які необхідні вузлам, намагаючись максимально зменшити їх локальність (аби якомога менші об’єми даних застосовувались лише одиничними вузлами системи), тим самим зменшуючи комунікативні витрати часу. Але такий аналіз не завжди можливий, наприклад для операцій з покажчиками дані можуть після прийому відтворюватись на кінцевих вузлах не зовсім коректно. Водночас цей підхід також орієнтований на застосування в гомогенних системах кластерного типу, які при цьому містять рівноцінні канали комунікації.

Очевидно, що обидва підходи хоч і забезпечують розширення технології розпаралелювання шляхом використання директив препроцессору, роблячи її доступною для використання в системах з локальною пам’яттю, але в жодному з них не враховується гетерогенність системи.

1.7.2. Технологія низькорівневої передачі повідомлень

При аналізі моделей паралельного програмування було визначено, що для систем з локальною пам’яттю єдиним доступним та природнім підходом наразі являється обмін низькорівневими повідомленнями між елементами.

Найпоширенішою технологією, яка дозволяє втілити дану модель, є MPI (англ. Message Passing Interface). З точки зору розробника реалізація MPI являє собою бібліотеку, орієнтовану перш за все на процедурну парадигму мови C, проте немає ніяких перешкод в застосуванні бібліотеки і в мові C++. Так само як і у випадку з OpenMP, існують адаптації бібліотеки до роботи з мовами C# та

Java, проте вони також не забезпечують повноцінного функціоналу та поступаються зручністю застосування вбудованим засобам, в основному через невідповідність прийнятих в цих мовах парадигм. Сама бібліотека повністю втілює всі аспекти моделі низькорівневої передачі повідомлень, описані в підрозділі 1.5.5.

Також варто відмітити декілька характерних особливостей, доданих в технології MPI. По-перше, буферизовані передачі, ідея яких полягає в копіюванні процесом-відправником даних до виділеного буферу, що дозволяє процесу призупинити обчислення лише на час цього копіювання, не відводячи свій час на обробку відправлення цих даних. Буфери також можуть створюватись і в процесах-приймачах, що створює передумови для відкладення передачі даних аж до моменту безпосередньої їх необхідності процесом-приймачем, що дозволяє вести значно ефективнішу роботу планувальників. Водночас головним недоліком буферизації є збільшення необхідних об’ємів пам’яті, аж до двох разів. Також буферизовані передачі мають строгий порядок відправки та прийому.

По-друге, повідомлення MPI можуть за необхідності виконувати обробку відправлених даних, що дозволяє адаптувати дані в залежності від особливостей роботи з пам’яттю операційної системи вузла-приймача.

Сучасні реалізації MPI мають також адаптації для випадків, коли вузли-приймачі знаходяться в рамках одного процесору, в цьому випадку не відбувається буферизація повідомлень та задіюються засоби операційної системи для розподілення пам’яті між процесами.

Основними недоліками MPI лишається низькорівневість (особливо прив’язка до процедурної парадигми), орієнтованість на гомогенні системи та канали зв’язку, а, відповідно, і відсутність оптимізованих для цього планувальників, відсутність можливості динамічної регуляції зерна паралелізму.

1.7.3. Технологія відвантажених обчислень

Технологія відвантажених обчислень втілює в собі реалізації задіяння акселераторів в обчислювальних процесах. Будь-яка система, яка містить акселератор автоматично повинна вважатись не тільки розподіленою, а й гетерогенною. Оскільки система вважається розподіленою, то єдина підходяща модель паралельного програмування для неї – відправка повідомлень. Проте нема жодних перешкод до розподілення даних між центральним процесором та акселератором, та реалізації обчислень на кожному з них окремо як на системах зі спільною пам’яттю.

Як вже було зазначено, найпоширенішим типом акселераторів наразі є графічні процесори. Перші графічні процесори компанії Nvidia, які підтримували технологію CUDA (від англ. Compute Unified Device Architecture, уніфікована обчислювальна архітектура пристрою - реалізація техніки GPGPU від цієї компанії) не відрізнялись дешевизною та доступністю. Проте досить швидко з’явились інші реалізації GPGPU, серед яких варто відмітити фреймворк OpenCL, який дозволяє програмувати загальні обчислення для GPU не лише від компанії Nvidia, а, наприклад, і компанії AMD. Також, окрім того, що OpenCL підтримує більше GPU ніж CUDA, зазвичай такі GPU дешевші та простіші, аж до звичайних користувацьких відеокарт.

Віднедавна технології Intel Threading Building Blocks та CUDA-MPI надають підтримку акселераторних обчислень в системах зі спільною пам’яттю (Intel TBB) або локальною (CUDA-MPI). Проте надається ця підтримка в рамках моделей з низьким рівнем абстракції, при цьому у випадку CUDA-MPI від цільових систем вимагається повнозв’язність та гомогенність на рівні каналів зв’язку, архітектури вузлів, продуктивності вузлів. Хоча гомогенність і не є безпосередньою умовою функціонування, але методи балансування навантаження, вбудовані в ці технології, не враховують можливу значну різницю в потужності складових елементів системи, як процесорів, так і акселераторів, вибираючи один із елементів та один із каналів (зазвичай –

найповільніший) як еталон, тим самим втрачається частина потужності продуктивніших вузлів.

1.7.4. Технологія відкладених обчислень

Класична модель обчислень передбачає, що останньою дією команди, процедури або функції є повернення результату (явне або неявне). Тобто, ланцюжок роботи програми передбачає, що користувач сам декларує прямий порядок виклику функцій, кожна з яких буде повертати результат, який може бути використаний наступними за порядком функціями.

Технологія відкладених обчислень передбачає інший порядок – рекурсивний. Ідея полягає в тому, що відкладена функція не буде виконуватись в програмі доти, допоки результат її роботи не буде явно запитано в коді. Це передбачає те, що виконання функції можна відкладати безкінечно довго, аж до критичного часу, тобто моменту першого запиту її результату в коді. Цей зручний інструмент дозволяє планувальникам (ручним або автоматичним планувальникам середовища виконання або операційної системи) розширити можливості до розподілу навантаження на системні ресурси, пам’ять та мережу, що створює значний заділ до покращення результатів роботи цих планувальників. Також відкладені обчислення є важливим елементом інтерпретованих мов програмування.

Найчастішим втіленням та застосуванням на практиці відкладених обчислень є відкладені функції передачі даних та функції зворотного виклику (коллбеки).

В загальному практика використання відкладених функцій та інструменти їх реалізації присутні в багатьох сучасних мовах програмування [17], їх концепція має загальну назву «ліниві обчислення» (англ. lazy evaluation). Ця концепція має три основні складові:

1. Відкладені обчислення (англ. deffered evaluation) – обчислення виконуються не за порядком їх опису в коді, а лише за безпосереднім їх запитом;
2. Мінімальні обчислення (англ. circuit evaluation) – обчислення виконуються рівно в тому обсязі, якого достатньо для отримання результату. Широко застосовуються для тонкої оптимізації на рівні мови. Наприклад в мовах Java та C# якщо в умовній конструкції з логічним оператором І (англ. AND) перший операнд false, то нема потреби перевіряти другий операнд, оскільки результат в будь-якому випадку буде false;
3. Інкрементальні обчислення (англ. incremental evaluation) – для отримання кожного окремого значення обчислення виконуються лише один раз. Їх реалізація найчастіше пов’язана із використанням чистих функцій, що є однією з засад функціональної парадигми програмування. Чистою є функція, яка має єдиний результат, значення якого визначається лише аргументами. Із цього слідує, що для кожної окремої можливої комбінації вхідних значень результат завжди буде незмінним, відповідно, при повторній подачі такої комбінації аргументів немає потреби вираховувати тіло функції, достатньо повернути попередній результат. Для реалізації цього повернення в сучасних мовах (як на внутрішньому, так і на прикладному рівнях) використовуються механізми мемоізації – збереження в пам’ять результатів попередніх викликів чистих функцій.

У разі, коли виконання функції зворотного виклику відбувається в окремому потоці або процесі, тобто асинхронно, з’являється простір до реалізації паралелізму в початково однопоточних мовах, таких як JavaScript, Node.js. Підтримку асинхронних паралельних функцій зворотного виклику реалізовано в мовах C#, Java, а від 2011 року додано і в стандартну бібліотеку Futures мови C++ [26].

1.7.5. Порівняння обраних технологій паралельного програмування

Зведемо порівняння особливостей застосування сучасних технологій паралельного програмування до єдиної таблиці (таблиця. 1.2).

Таблиця 1.2

Порівняльна таблиця особливостей сучасних технологій паралельного програмування

Технологія	Цільова ПКС	Режим доступу до даних
Директиви препроцесору (OpenMP)	ПКС з СП	Копіювання автоматичне відповідно до вказівок; додаткові вказівки для взаємного виключення; лише спільна пам’ять.
Низькорівневий інтерфейс передачі повідомлень (MPI)	ПКС з ЛП	Обов’язкове явне копіювання даних між вузлами; можливість програмованої буферизації під час передачі.
Інтерфейс програмування графічних акселераторів (Nvidia CUDA, OpenCL)	ПКС з СП та акселераторами	Обов’язкове явне копіювання даних в пам’ять акселератора. Багаторівнева ієрархія пам’яті.
Низькорівневий інтерфейс передачі повідомлень з підтримкою графічних акселераторів (CUDA-MPI)	ПКС з ЛП та акселераторами	Те ж саме, що попереднє, та додатково можливість явного копіювання у пам’ять акселератора в іншому вузлі.
Технологія Intel Threading Building Blocks	ПКС з СП та застосуванням різних акселераторів	Автоматичне копіювання на акселератор та взаємне виключення в екземплярах спеціально структурованих типів; лише спільна пам’ять.
Стандартна бібліотека Futures мови C++ , стандартні засоби мов C#, Java, Node.js	ПКС з СП	Можливість відкладеного обчислення та доступу (за безпосереднім запитом), зокрема паралельного. Лише спільна пам’ять.

Висновки до розділу 1

1. Паралельні комп'ютерні системи зі спільною пам'яттю мають цілий ряд переваг, однак можливості їх розширення принципово обмежені зверху, тому сучасні комп’ютерні системи, орієнтовані на роботу з високонавантаженими обчисленнями бажано проектувати як системи з локальною (розподіленою) пам’яттю;
2. Але в той же час сучасні програмні комплекси, які забезпечують роботу систем з локальною пам’яттю декларують можливість автоматичного переходу системи до моделей організації паралельних процесів, орієнтованих на системи зі спільною пам’яттю в тих випадках, коли виконання доходить до кінцевих вузлів, на яких з’являються можливості до такого переходу;
3. При проектуванні сучасних систем та засобів паралельного програмування спостерігається тенденція авторів приховувати всю системну реалізацію низького рівня, надаючи розробникам-користувачам можливості оперувати більш високорівневими абстракціями, які дозволяють не вимагати від останніх адаптувати прикладні задачі та бізнес-логіку до особливостей архітектури цільової системи, тим самим даючи їм можливість зосередитись на реалізації прикладної задачі, додатково забезпечуючи уникнення потенційних помилок на рівні паралельної системи;
4. Наразі навіть при організації розподілених паралельних комп’ютерних систем доцільним може бути застосування технологій відкладених обчислень, особливо для більш ефективного використання комунікативного середовища системи;
5. Наразі стрімкої популярності набирають гетерогенні паралельні системи, в яких використовуються акселератори, і, відповідно, технології відвантажених обчислень.

6. Як видно із класифікації, наведеної в таблиці 1.2, сучасні засоби паралельного програмування для систем із локальною пам’яттю орієнтовані на гомогенні системи, перш за все на повнозв’язні кластерні. Навіть ті, які підтримують відвантажені обчислення, передбачають використання гомогенних акселераторів. При цьому в них не передбачається застосування технологій відкладених обчислень;
7. Проблема описаної в попередньому підпункті підтримки полягає не в тому, що система не буде працювати у разі гетерогенності її вузлів, а в тому, що балансування навантаження, яке є невідмінною частиною роботи будь-якої паралельної чи розподіленої системи, буде виконуватись не оптимально, відповідно коефіцієнти прискорення та ефективності вузлів та елементів вузлів системи будуть нижчими, ніж могли би бути.

Отже, з’являється проблема відсутності методу балансування навантаження в розподілених комп’ютерних системах, при роботі якого одночасно враховувались би такі фактори, як:

1. Гетерогенність кінцевих вузлів системи;
2. Наявність неоднорідних акселераторів в цих вузлах;
3. Потенційну неповнозв’язність системи;
4. Гетерогенність комунікативного середовища системи.

Тому необхідно розробити такий метод, і при цьому врахувати ряд наступних факторів:

1. Для збільшення продуктивності розробників паралельного програмного забезпечення в рамках методу необхідно дозволити їм зосередитись на розробці логіки задачі, не наближаючи її до рівня системи, і також цим самим зменшити кількість помилок системного рівня, а також створити передумови до автоматичної оптимізації, варто використовувати такі моделі паралельного програмування, які забезпечують високі рівні абстракції.
2. Хоч модель із автоматичним створенням потоків та модель із відсутніми блокуванням є найбільш абстрактними та найперспективнішими, але вони

поки що мають обмеження на класи задач, які можуть бути реалізовані в таких системах, і самі являються складними в реалізації, перш за все через відсутність поняття даних як елементу паралельної задачі в таких системах. Оптимальніше наразі зупинити вибір на моделі із динамічним створенням потоків, увівши до нього деякі особливості автоматичного створення потоків, такі як автоматичний поділ задач та даних за заданим розробником шаблоном (наприклад як це робиться в реалізації механізму Fork-Join (RecursiveTask, RecursiveAction) покету java.util.concurrent мови Java) [27] та автоматичну регуляцію зерна паралелізму.

3. Але у цієї моделі є недолік – орієнтованість на ПКС з ЛП. Тому необхідно скомбінувати її із моделлю низькорівневої передачі повідомлень, але, щоб забезпечити прийнятий високий рівень абстракції, покласти підготовку даних, організацію всіх передач та керування ними на планувальник.
4. Також необхідно використати одну з реалізацій технології відвантажених обчислень для забезпечення підтримки ПКС з акселераторами.

Так само як згідно з формулами (1) та (2) критичні ділянки паралельних програм в значній мірі обмежують продуктивність ПКС з СП, так передачі повідомлень між вузлами ведуть до простоїв та, відповідно, зменшення ефективності ПКС з ЛП. Отже, в розроблюваному методі бажано також:

1. Оптимізувати пересилки повідомлень шляхом зменшення їх розміру. Недоцільно передавати всі дані всім вузлам. Очевидно, що якщо наявні дані векторного типу, які згідно з паралельною реалізацією задачі можна розділити, то варто це зробити. Та навіть у випадку відсутності таких даних може бути недоцільно масово розсилати всі спільні неподільні дані всім вузлам, якщо деякі із них не потребують деяких із цих даних.
2. Оптимізувати пересилки шляхом використання технології відкладених обчислень. Згідно з дослідженням [17], у більшості випадків це створює заділ для більш ефективного планування обчислень та балансування навантаження, що веде до підвищення швидкодії програм та коефіцієнтів ефективності систем.

РОЗДІЛ 2
РОЗРОБКА МЕТОДУ БАЛАНСУВАННЯ НАВАНТАЖЕННЯ В РГКС

В даному розділі наведено опис запропонованого методу балансування навантаження в розподілених гетерогенних комп'ютерних системах, та пов'язаного з ним методу організації таких систем. Приведені вимоги до описаного методу та системи в цілому, розглянуто різні режими роботи компонентів та методи моделювання цих режимів. Наведено детальні теоретичні описи всіх використаних при цьому алгоритмів.

2.1. Апаратна складова РГКС

Перш ніж приступити до опису алгоритму балансування навантаження, необхідно провести опис системи, прийнятих в ній моделей і технологій, режимів її роботи.

Цільова розподілена система являє собою набір вузлів та зв'язків між ними. Вузол являє собою паралельну комп'ютерну систему зі спільною або з локальною пам'яттю. Тип пам'яті визначається тим, чи наявний акселератор в даному вузлі. Якщо акселератор наявний, то вузол автоматично вважається системою з локальною пам'яттю, складовими якої є акселератор та центральний процесорний елемент, який в свою чергу (разом з оперативною пам'яттю вузла) утворює систему зі спільною пам'яттю. Таким вузол вважається і за відсутності в ньому акселератора. Тобто, гарантовано в кожному вузлі на найнижчому рівні організації присутня паралельна комп'ютерна система зі спільною пам'яттю.

Оскільки в системі передбачається наявність вузлів з акселераторами, то зв'язки можуть бути двох типів:

1. На рівні розподіленої системи – будь-які доступні комунікативні канали в рамках локальної та/або глобальної комп'ютерної мережі;

2. На рівні вузла – інтерфейс взаємодії акселератора з материнською платою комп’ютерної системи.

2.2. Програмна складова РГКС

Оскільки основним завданням системи є організація паралельного вирішення задач, то необхідно відштовхуватись від способів її представлення в системі. Найбажанішим звісно є організація найвищого рівня абстракції, при якому вимагається представлення лише математичного апарату, із покладенням на систему обов’язку реалізації виділення паралельних секцій та їх виконання. Проте, існує багато задач, для яких не можливо виконати математичний алгоритм в паралельному режимі одразу, але можливо при ручному внесенні деяких змін. Тому варто зупинитись на найпопулярнішому наразі підході в високого рівня абстракції, в рамках якого вимагається представлення математичного вирішення задачі із мінімальним ручним виділенням бажаних до паралельного виконання секцій. Доцільно прийняти схему програмного інтерфейсу, запропоновану професором Стіренко [17] та дещо її модифікувати. Модифікація полягає в розширенні спектру операцій, а саме:

1. Загальне розділення даних;
2. Загальний збір даних;
3. Загальне розділення задачі;
4. Часткове розділення даних;
5. Частковий збір даних;
6. Часткове розділення задачі.

При цьому перші три функції є явними (тобто доступними користувачу), а останні три функції є службовими, виконання яких є важливим етапом програми, але приховане від користувача.

Операції з даними при цьому доцільно модифікувати так, як це робиться в технологіях OpenMP та MPI на рівнях паралельної системи зі спільною пам’яттю та локальною пам’яттю відповідно. При цьому, у зв’язку з деякими

особливостями роботи алгоритму оцінки задачі (описані нижче), з'являються можливості до автоматичної обробки доступності даних, а не лише в ручному режимі.

Такий інтерфейс є достатнім для ефективної організації паралельного виконання абсолютної більшості прикладних математичних задач. Автори дослідження [28] приводять схеми роботи, до яких приводяться більшість сучасних паралельних задач (рис. 2.1). Це підтверджується також фактом, що до 80% роботи програм займає виконання циклічних та ітеративних конструкцій [25], які також підпадають під наведені схеми.

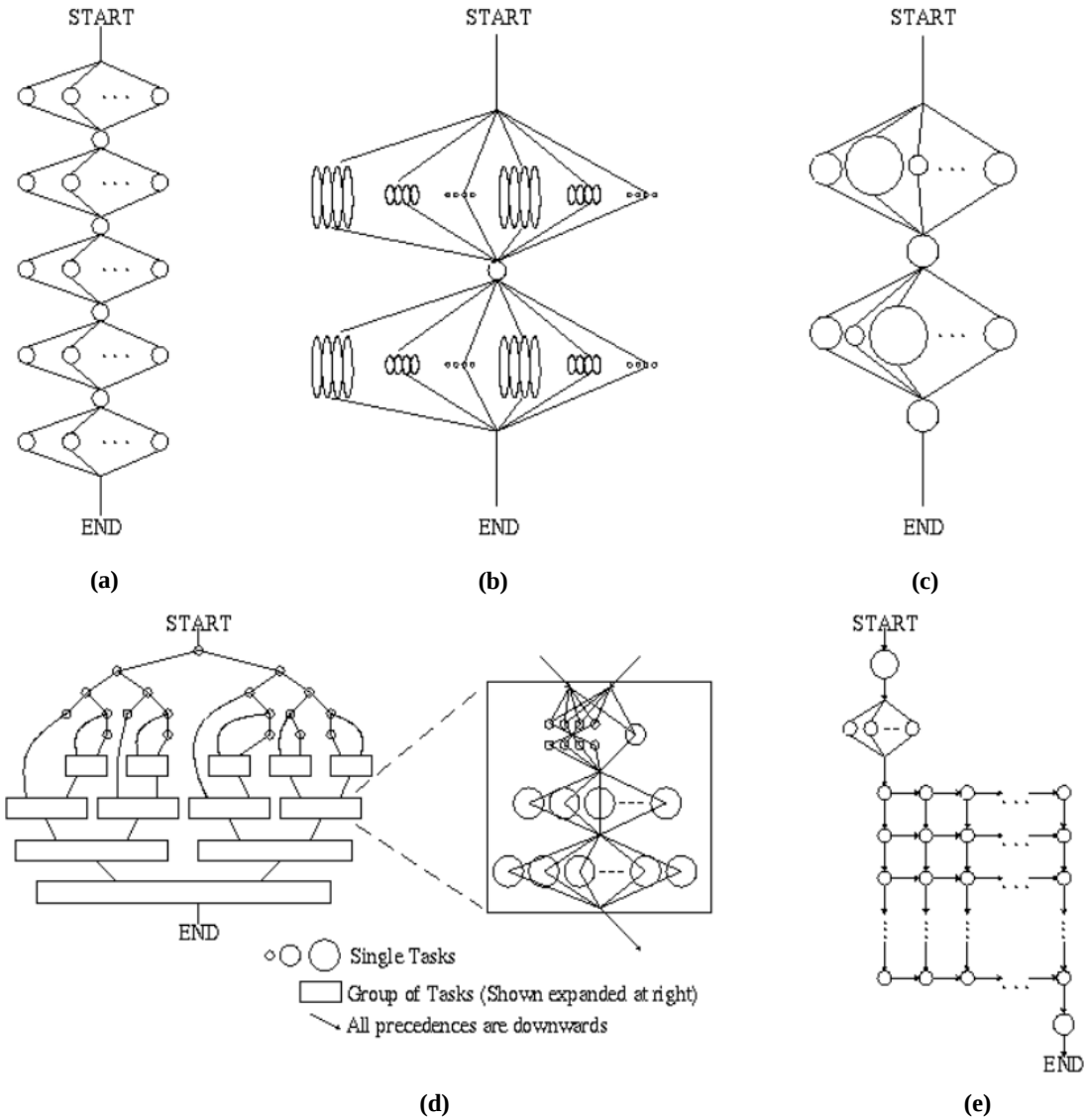

Рис. 2.1. Приклади найрозповсюдженіших схем роботи (графів паралельних задач) паралельних програм

Проте варто відмітити, що і цей інтерфейс трохи зниженого рівня абстракції не покриває всі можливі види задач. Наприклад при роботі розподіленої системи в режимі розподіленого серверу схема роботи буде принципово відрізнятись. Тому доцільно залишити можливість використання низькорівневих примітивів.

Загалом вхідні паралельні задачі в сучасних загальноприйнятих підходах до програмування представляються відповідно до одного з двох наступних паттернів паралелізму:

1. “Data-driven parallelism”. Основна ідея цього паттерну полягає в тому, що кожен обчислювач в системі виконує одну і ту ж послідовність операцій над своєю частиною вхідних даних, не залежною від інших, отримані результати кожного обчислювача в кінці збираються та формуються в єдиний фінальний результат.
2. “Task-driven parallelism”. Основна ідея цього паттерну полягає в протилежному – в системі між обчислювачами курсує один і той же набір вхідних даних, і кожен обчислювач виконує над ним свої, відмінні від інших набори операцій. Результат роботи фінального обчислювача і є остаточним результатом.

Майже всі наведені на рисунку 2.1 класифікації належать до «Data-driven parallelism». Варто відмітити конструкцію, яка приводить нас до ще двох важливих понять - регулярного та нерегулярного паралелізму (рис.2.1, d).

Регулярним можна вважати такий паралелізм, при якому кожен потік виконує абсолютно однаковий набір інструкцій. Якщо ж з'являється ситуація, коли всі потоки, або деякі із них виконують інший набір інструкцій (який може бути як принципово іншим, так і умовно, тобто залежати від конкретних зовнішніх факторів, вхідних умов, ідентифікатору потоку, тощо), то такий паралелізм вважається нерегулярним.

Відповідно, «Task-driven parallelism» завжди принципово вважається нерегулярним, а «Data-driven parallelism» може бути як регулярним, так і ні.

Тобто конструкція (рис.2.1, e) відповідає нерегулярному «Data-driven parallelism».

Оскільки, як було зазначено, абсолютна більшість паралельних математичних задач підпадають під схему «Data-driven parallelism», то основна увага в описаній системі надається саме регулярному та умовно-нерегулярному (коли нерегулярність операцій є тенденцією, а не винятком (рис.2.1, b,c)) його режимах.

2.4. Алгоритм оцінки вхідної задачі

Оцінка вхідної задачі необхідна з двох причин:

1. Для більш коректної оцінки продуктивності вузлів, характерної саме для цієї задачі;
2. Для формування автоматичної обробки спільних для процесів даних.

В обох випадках для цього перш за все необхідно засобами мови реалізації побудувати абстрактне синтаксичне дерево вхідної задачі за правилами даної мови.

Абстрактним синтаксичним деревом задачі називається скінченна множина вершин та зв’язків між ними, які утворюють між собою позначену орієнтовану деревовидну структуру. В цій структурі внутрішні вершини відповідають операторам мови програмування, а листя являє собою операнди. Такі структури використовуються на етапі парсингу програмного коду. Вони необхідні для проміжного представлення програми між конкретним синтаксичним деревом і структурою даних, яка потім використовується як внутрішнє представлення компілятора або інтерпретатора програми для оптимізації і генерації коду.

Для прикладу візьмемо наступний псевдокод обчислення алгоритму Евкліда для знаходження найбільшого спільного дільника двох цілих чисел:

```
1. while b ≠ 0
2.     if a > b
```

3. a := a − b
4. else
5. b := b − a
6. return a

Абстрактне синтаксичне дерево цього коду матиме загальний вигляд, приведений на рисунку 2.2.

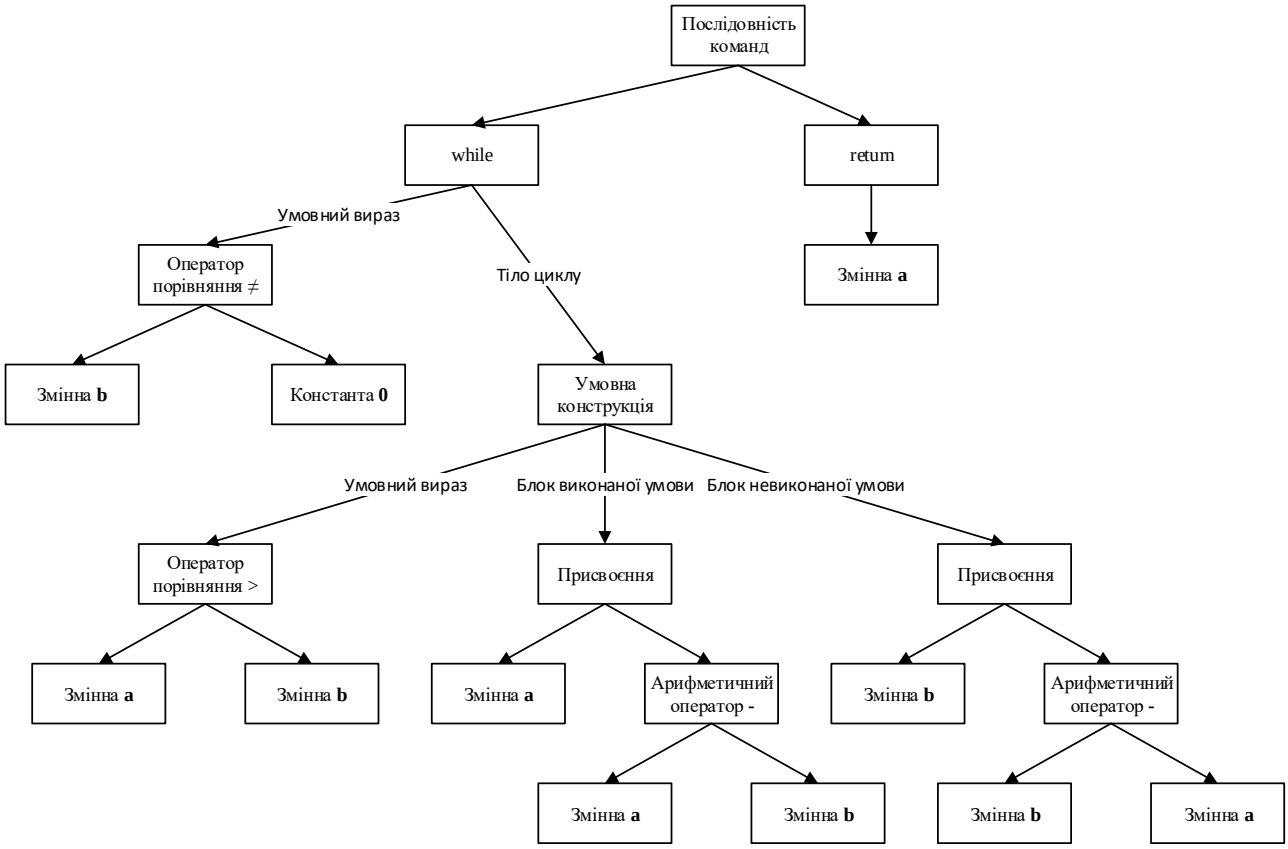

Рис. 2.2. Узагальнене абстрактне синтаксичне дерево для наведеного вище алгоритму Евкліда

Оскільки передбачається відповідність задачі перш за все паттерну «Data-driven parallelism», то можна стверджувати, що кожен кінцевий потік буде в ідеальному випадку виконувати однакові дії. Тоді, з врахуванням того, що в кожній сучасній комп’ютерній системі виконання будь-якого алгоритму зведеться до виконання послідовності елементарних арифметичних операцій [29], можна провести орієнтовну оцінку об'ємності таких операцій, присутніх в задачі. Для цього:

1. В графі задач за методом обходу в глибину виявимо спершу наявність циклічних конструкцій та вирахуємо їх потужність (максимально можливу кількість задекларованих повних ітерацій, тобто без врахувань переривань). У випадку, коли межі циклічної конструкції залежать від об'єму певного набору даних ми можемо використати показник повного об'єму цих даних, тому що на цьому етапі програми він вже буде відомий (опис цього приведено нижче). За результатами цього кроку ми маємо набір значень $I_1..I_N$, де I – потужність конкретного циклу, N – порядковий номер циклу.
2. Далі ведеться знову пошук в глибину, але лише від вершин циклічних конструкцій, тобто проходяться цикли від 1 до N. Ведеться підрахунок значень C_{SUM}, C_{SUB}, C_{MUL}, C_{DIV}, C_M – кількості елементарних операцій додавання, віднімання, множення, ділення і присвоєння відповідно, які зустрілись в цьому циклі. В кінці кожного обходу кожне значення домножується на відповідний цьому циклу показник потужності I;
3. Модифікуємо отримані значення відповідно до наявності позациклових елементарних операцій, збільшивши відповідні значення C_{SUM}, C_{SUB}, C_{MUL}, C_{DIV}, C_M на кількість виявлених відповідних операцій.

Отримані в результаті фінальні значення показників C_{SUM}, C_{SUB}, C_{MUL}, C_{DIV}, C_M будуть використані в алгоритмах роботи інших компонентів, описаних в наступних підрозділах нижче.

За абстрактним синтаксичним деревом також можна визначити наявність спільних даних. Оскільки операції розділення даних та розділення задачі декларуються окремими функціями в коді, то можна стверджувати, що всі дані, які одночасно знаходяться в дереві в тій гілці, яка відповідає за задачу, яка підлягає розділенню, та гілках, які відповідають за попередні конструкції (тобто знаходяться в графі на одному й тому ж рівні ліворуч) потребують копіювання. Ті ж дані, які крім цього знаходяться ще в тих гілках, виконання яких передбачається наступним після розділення (тобто які знаходяться на

одному й тому ж графу рівні праворуч) вимагають організації потокобезпечної обробки.

2.5. Алгоритм ініціалізації системи

Як було сказано вище, система являє собою набір розподілених слабозв'язних вузлів. В системі передбачається початкова рівноправність всіх вузлів, тобто будь-який вузол може виступити ініціатором обчислень. Від цієї операції ініціалізації вузлом обчислень і починається ініціалізація системи.

Оскільки система може динамічно змінюватись шляхом додавання чи віднімання вузлів, то жоден вузол спершу не має повної інформації про всю систему. Перед початком обчислення вузол має лише дані про ті вузли, яким він він може делегувати обчислення. Ці дані на кожному вузлі зібрані в його власний конфігураційний файл, і являють собою набори значень $Z_i=\{IP_i, PORT_i, CPU_i, ACC_i\}$, де i – порядковий номер під'єднаного вузла, IP_i – адреса вузла i, $PORT_i$ – виділений програмі порт на вузлі i, CPU_i – відсоток потужностей, який може бути програмою використаний на цьому вузлі на центральному процесорнорму елементі, ACC_i – відсоток потужностей, який може бути програмою використаний на цьому вузлі на акселераторі.

Згідно з цими даними відбувається логістична перебудова системи таким чином, щоб вона являла собою орієнтований ациклічний граф-дерево, в якому коренем є вершина-ініціатор, на першому рівні розташовуються вершини, яким безпосередньо може делегувати обчислення ініціатор, на наступному ті вершини, яким можуть делегувати обчислення вершини першого рівня, і так далі. Акселератори вважаються окремими вузлами, поєднаними лише із вузлами, в яких вони встановлені. Схематичний приклад такого графу для системи з 10 різноманітних за складом вузлів представлено на рисунку 2.3.

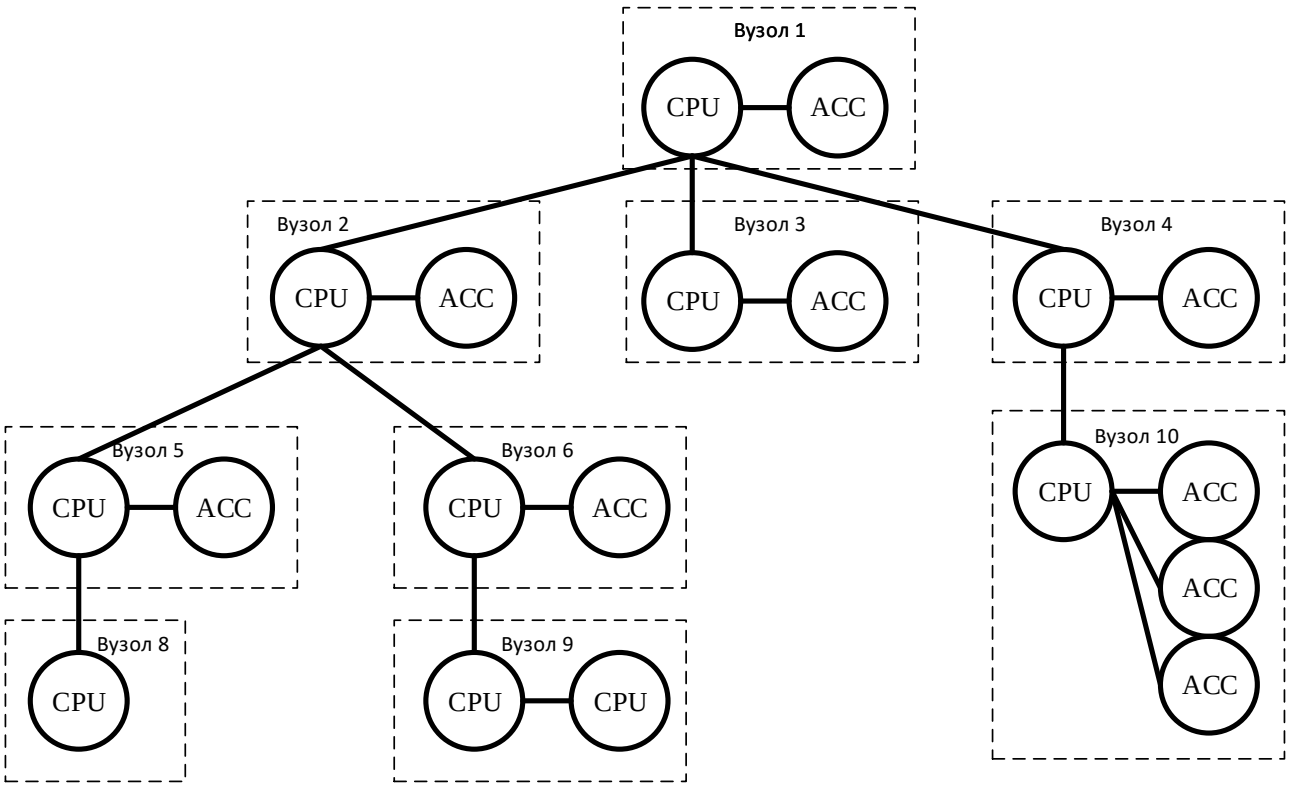

Рис. 2.3. Схематичний приклад графу для системи з 10 різноманітних за складом вузлів

На наступному кроці необхідно зважити отриманий граф-дерево відносними показниками продуктивності вузлів та елементів вузлів.

2.5.1. Зважування ребер графу системи

Для зважування ребер необхідно спершу провести оцінку абсолютної пропускної здатності каналів зв’язку між вузлами, а також, за наявності на вузлі акселераторів, то і інтерфейсів зв’язку між процесорним елементом вузла та акселераторами.

Для абсолютної оцінки пропускної здатності інтерфейсів зв’язку з акселераторами достатньо провести тестові відправки пакетів даних фіксованого розміру (але які містять різні типи даних) до них та прийому пакетів даних фіксованого розміру від них, і заміряти відповідні показники часу. Тобто, пропускна здатність інтерфейсу W_I буде визначатись за наступною формулою (3):

$$W_I = \frac{2 * S}{\Delta T_{SEND} + \Delta T_{RECV}}\ , \tag{3}$$

де ΔT_{SEND} – різниця між часом початку та закінчення процесу відправки даних, ΔT_{RECV} - різниця між часом початку та закінчення процесу отримання даних, S – обсяг відправлених даних.

Для оцінки абсолютної пропускної здатності каналів зв'язку між вузлами можна скористатись стандартною утилітою traceroute. Часові дані, отримані за рахунок її використання, в меншій мірі піддаються впливу відсутності гарантій єдиного маршруту слідування пакету, ніж ті, що отримані в результаті використання утиліти ping. Пропускна здатність комунікаційного каналу між двома вузлами W_C буде визначатись за наступною формулою (4):

$$W_C = \sum_{i=1}^{N} \left(\frac{S_i}{\sum_{j=1}^{M} Tj} \right), \tag{4}$$

де N – кількість запусків утиліти traceroute, M – кількість проміжних вузлів на шляху слідування пакету на поточному запуску, T – час проходження пакетом від одного вузла маршруту до іншого, S – розмір відправленого на поточному запуску пакету.

Отримані абсолютні показники пропускної здатності каналів зв’язку та інтерфейсів необхідно окремо перевести у відносні як і показники продуктивності елементів, але зважування в цьому випадку вестиметься не відносно сумарного показника, а відносно найкращого показника. Тобто, за формулою складних відсотків відносний показник пропускної здатності каналу W_{CR} визначатиметься формулою (5):

$$W_{CR} = \frac{W_C * 100}{W_{CB}}\ , \tag{5}$$

де W_C – абсолютний показник пропускної здатності каналу, W_{CB} – абсолютний показник продуктивності найшвидшого із каналів в системі.

Відповідно, відносний показник пропускної здатності інтерфейсу взаємодії з акселератором визначатиметься за формулою (6):

$$W_{IR} = \frac{W_I * 100}{W_{IB}} \ , \tag{6}$$

де W_I – абсолютний показник пропускної здатності інтерфейсу, W_{IB} – абсолютний показник продуктивності найшвидшого із інтерфейсів в системі.

В результаті отримані відносні показники продуктивності каналів і інтерфейсів і будуть відповідати вагам відповідних ребер графу системи.

2.5.2. Зважування вершин графу системи

Для зважування вершин спершу необхідно отримати абсолютні показники продуктивності елементів вузлів. Для цього можна запропонувати наступний підхід:

Спершу необхідно отримати набір показників T_{SUM}, T_{SUB}, T_{MUL}, T_{DIV}, T_M – середній час виконання елементом операцій додавання, віднімання, множення, ділення і присвоєння відповідно. Для цього необхідно на кожному елементі (як центральних процесорних елементах, так і акселераторах) виконати деяку завчасно задану кількість кожної зі згаданих операцій над певними завчасно згенерованими випадковими різнотипними даними та провести заміри часу який було витрачено на кожну операцію. Цю підпрограму варто виконувати з попередньою вказівкою компілятору не проводити оптимізацію, аби вона не впливала на результат.

Після отримання середніх часових показників за формулою (7) можна обчислити W_{EA} - абсолютний показник продуктивності елементу вузла системи:

$$W_{EA} = \frac{P * \left(\frac{C_{SUM}}{T_{SUM}} + \frac{C_{SUB}}{T_{SUB}} + \frac{C_{MUL}}{T_{MUL}} + \frac{C_{DIV}}{T_{DIV}} + \frac{C_M}{T_M}\right)}{\Delta L * \Delta t * \Delta M} \ , \tag{7}$$

де T_{SUM}, T_{SUB}, T_{MUL}, T_{DIV}, T_M – середній час виконання елементом операцій додавання, віднімання, множення, ділення і присвоєння відповідно, C_{SUM}, C_{SUB}, C_{MUL}, C_{DIV}, C_M – кількості в задачі елементарних операцій додавання, віднімання, множення, ділення і присвоєння відповідно, P – кількість ядер (або

потоків) цього елементу, ΔL – різниця початкового та кінцевого показників завантаженості елементу, Δt – різниця початкового та кінцевого показників температури елементу, ΔM – різниця початкового та кінцевого показників кількості хеш-промахів елементу (для акселераторів не враховується).

Далі необхідно провести переведення абсолютних показників продуктивності у відносні відсоткові. При цьому необхідно врахувати, що сумарно в системі час обробки підзадачі на вузлі визначається не лише часом безпосереднім часом виконання обчислень, а й часом простою цього вузла в очікуванні передачі даних. Тому доцільно модифікувати значення абсолютної продуктивності елементу відповідно до значення відносної пропускної здатності каналу зв'язку, який веде до цього елементу.

Сума відносних показників повинна становити 100%. Відповідно, за правилом складних відсотків, визначаємо W_{ER} - відносний показник продуктивності кожного елементу за формулою (8):

$$W_{ER} = \frac{\sum_{i=1}^{N} \left(W_{EA_i} * W_{R_i}\right)}{W_{EA^*} W_R}, \tag{8}$$

де N – сумарна кількість елементів в системі, W_{EA} – абсолютний показник продуктивності елементу, W_R – відносний показник пропускної здатності каналу зв'язку або інтерфейсу взаємодії з елементом.

В результаті отримані відносні показники продуктивності елементів системи будуть відповідати вагам відповідних вершин графу системи. Тому остаточно на цьому кроці маємо логістичну карту системи для конкретного обчислення. Фінальний (враховано тільки вагу вершин, в яку вже включено вагу ребер) умовний приклад такої карти для схеми із структурою відповідною структурі на рисунку 2.3 зображено на рисунку 2.4.

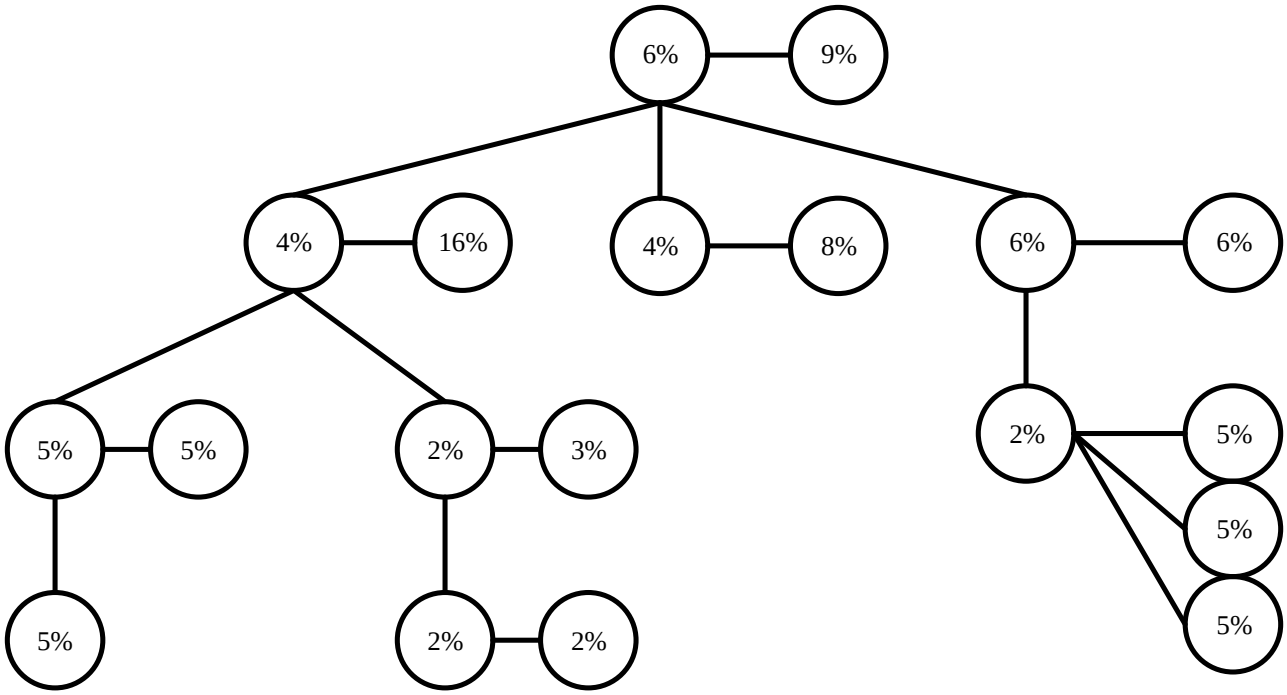

Рис. 2.4. Схематичний приклад зваженого графу (вага вузлів враховує вагу ребер) для системи зі структурою, аналогічною структурі з рис.2.3

Після виконання цього кроку система готова до початку планування та виконання безпосередніх обчислень, якими від самого початку цілком і повністю керують балансувальники навантаження на різних рівнях.

2.6. Алгоритм балансування навантаження

В рамках сучасних підходів до організації комп’ютерних систем розрізняються два основні типи балансування навантаження в системі: статичне та динамічне.

Балансувальник, який реалізовує статичне балансування виконує основну свою роботу перед безпосереднім запуском паралельної програми на виконання. Базуючись на даних про систему (кількість елементів, їх продуктивності, зв’язки між елементами, пропускна здатність каналів) та даних про задачу (кількість підзадач, наявність зв’язків між ними, об’ємність і інтенсивність пересилок між підзадачами, наявність спільних ресурсів) та згідно з певним прийнятим алгоритмом він проводить пошук оптимального розподілу підзадач по наявних обчислювальних елементах згідно з обраним критерієм

ефективності (зазвичай намагаються досягнути мінімізації часу роботи або максимізації коефіцієнту ефективності). Далі роль балансувальника зводиться лише в почерговому запуску задач на виконання, згідно з розробленим планом, без відступів від нього.

Задача балансування є NP-повною, тому отримане балансувальником рішення є лише наближенням до ідеального. При цьому зазвичай чим ефективніше таке наближення, тим більше часу необхідно витратити на його пошук за складними алгоритмами, що збільшує сумарний час роботи системи, що є значним недоліком. Також недоліком статичного балансування є те, що при обробці підзадач на вузлах балансувальник не враховує поточні показники стану цих вузлів, а також стану обчислень. До того ж, рішення, отримане статичним балансувальником буде нівельовано у випадку, якщо під час обчислень якийсь із вузлів системи від'єднається, що зведе нанівець всі попередні плани обчислень, і викличе необхідність повторного ребалансування задач, які лишились невиконаними.

Проте відсутність моніторингу системи під час обчислень водночас є і перевагою статичного балансувальника, оскільки система не витрачає ресурси на моніторинг та динамічне ребалансування, що дозволяє їй задіяти всі потужності лише для обчислень. Особливо помітно це при балансуванні на рівні розподіленої системи (рис 1.1), тому що показники моніторингу стану вузлів повинні курсувати через мережеві канали, які є значно повільнішими, ніж внутрішні канали кінцевих вузлів.

На противагу статичному балансувальнику, динамічний балансувальник майже не витрачає часу на підготовку попереднього оптимального рішення, а одразу приступає до назначення підзадач обчислювальним вузлам та організації передач. В процесі його роботи ведеться постійний моніторинг показників стану системи та черг підзадач, на основі яких приймаються рішення про розподіли між вузлами наступних підзадач.

Серед його переваг можна відмітити, що цей вид балансування є набагато більш гнучкішим, і дозволяє максимально задіювати всі системні ресурси.

Зазвичай рішення, знайдені динамічним балансувальником виявляються більш якісними, ніж знайдені статичним. Також при динамічному балансуванні збільшується стресостійкість системи та знижуються витрати на обробку стресових ситуацій – у випадку втрати зв’язку з одним з вузлів балансувальник просто повторно додасть в чергу підзадачі, виконання яких доручалось втраченому вузлу.

Недоліком, відповідно, є те, що в системі необхідно виділяти певну частину ресурсів (як обчислювальних, так і комунікативних) для роботи процесу моніторингу вузлів.

Також для ефективного вирішення задачі балансування, з врахуванням апаратного опису, доцільно модифікувати схему організації РКС (рис. 1.1), прибравши абстрактний рівень кластерної системи (оскільки декларується що кожен вузол також може бути розподіленою неповнозв’язною системою, яка є більш широким поняттям за кластерну та розширює можливості схеми) та виділивши новий абстрактний рівень – рівень паралельної системи, та відповідний йому рівень планувальника паралельної системи. Необхідність цієї модифікації та переваги, які вона надає, випливають із подальшого опису процесу балансування, який пропонується в системі.

Відповідно до введених змін в схему роботи РКС маємо модифікацію схеми із рисунку 1.1, зображену на рисунку 2.5.

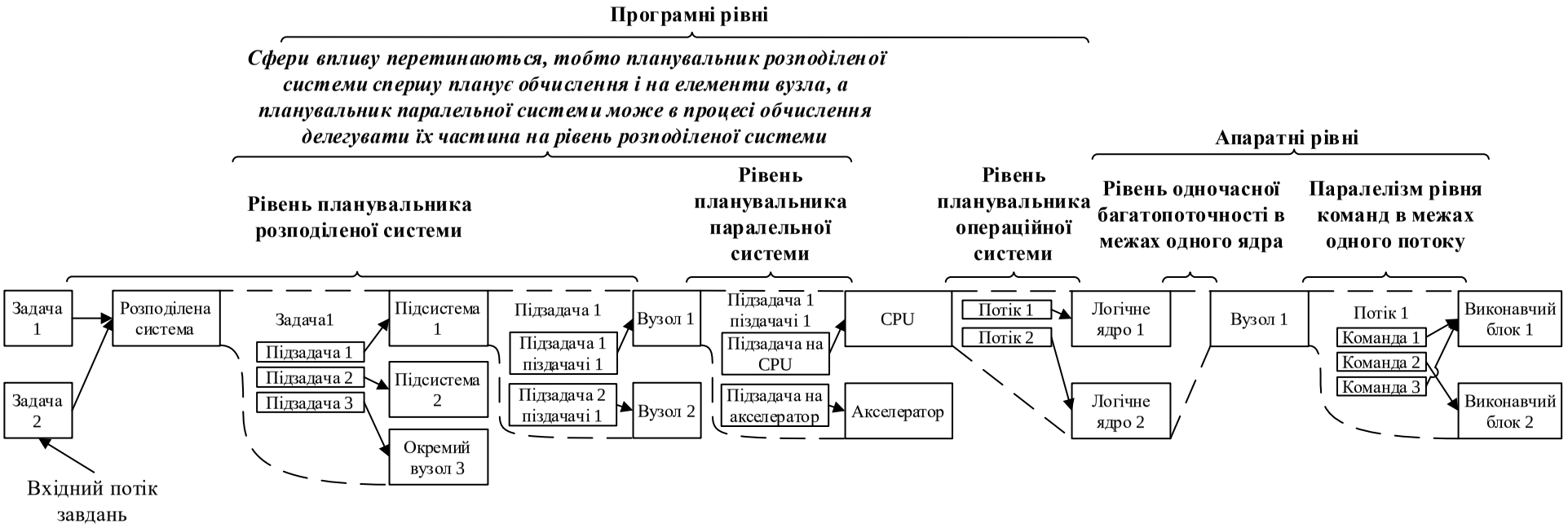

Рис. 2.5. Модифікована схема процесу обробки задач в розподілених гетерогенних комп’ютерних системах

В роботі пропонується застосувати одночасно і статичне і динамічне балансування на різних етапах обробки завдання на різних рівнях системи (рівні розподіленої системи та рівні паралельної системи, а також між ними) відповідно до схеми, зображеної на рисунку 2.5.

2.6.1. Статичне балансування

Як було відмічено вище, основними недоліками статичного балансування є:

1. Наближеність отриманого рішення;
2. Додаткові витрати часу на етапі підготовки;
3. Низька стресостійкість.

Оскільки в системі передбачається виконання паралельних програм, в яких паралелізм виділений на основі послідовних паралельних секцій (рис. 2.1) і підпадає під паттерн «Data-driven parallelism», то згідно з проведеними раніше дослідженнями [14-15] можна стверджувати, що якщо на рівні розподіленої системи провести початкове статичне балансування, в якому відсоток використання вузла залежатиме від об’єму даних, обробка яких очікується на даному вузлі, а об’єм даних визначатиметься відносною продуктивністю цього вузла на рівні системи, то можна отримати достатньо ефективне попереднє наближення вирішення задачі балансування навантаження за невеликий проміжок часу.

Відповідно до опису основних функцій для розпаралелювання програми, приведеного в підрозділі 2.3, виконуються перші три із описаних функцій, а саме «Загальне розділення даних», «Загальний збір даних», «Загальне розділення задачі», при чому їх застосування нерозривно пов'язане між собою.

На вузлі-ініціаторі обчислення виконується спершу функція загального розділення даних. Аргументами цієї функції є дані, які будуть розділені між вузлами. При цьому очевидно, що поділ відбувається лише відмічених даних, до того ж таких, які являють собою векторну структуру в пам’яті (наприклад

масиви, матриці, списки, символи рядків, рядки документів, пікселі зображення, тощо), якщо ж тип даних неподільний, то ці дані просто копіюються.

Розподіл векторних даних відбувається за основі показників ваги вершин графу-карти системи, отриманих на попередньому кроці. Ваги ребер не враховуються на цьому кроці, тому що вони враховані вже на етапі формування відносних ваг вершин (8). Тому розмір частини даних S_P для кожного вузла визначається за наступною формулою (9):

$$S_P = \lceil S * W \rceil, \tag{9}$$

де S – розмір векторної структури даних, яка підлягає поділу, W – вага (у відсотках) вузла на карті системи.

В той же час в системі, відповідно до кількості вузлів, створюється набір динамічних об’єктів, клас яких може додатково (але не обов’язково) наслідуватися від класу паралельної задачі або потоку. Ці об’єкти мають перш за все поля-ідентифікатори (які відповідають ідентифікаторам вузлів в системі), а також динамічні поля, в які додаються дані, які необхідно надіслати вузлам.

Кожен вхідний вектор, який підлягає надсиланню із розбиттям при цьому становить відповідну частину розміром S_P загального вектору в програмі, тобто від вхідного вектору почергово відрізаються частини, розмірність яких відповідає черговому значенню S_P із списку, отриманого на попередньому кроці.

Наступною загальною функцією є «Загальне розділення задачі». Ця функція приймає функцію, виконання якої розробник-користувач бажає здійснити в паралельному режимі, функцію, в якій міститься код, виконання якого передбачається на акселераторі, а також ідентифікатор задачі. Все це копіюється у відповідні динамічні об’єкти із попереднього списку, при цьому кожен цей об’єкт вже має ідентифікатор, який і передається паралельній функції та акселераторній функці. Це необхідно в тому числі для того, що якщо розробник передбачає використання розподілених даних в своїй програмі, то він мав змогу єдиним кодом здійснити обробку цих даних, в залежності від ідентифікатора вузла (якщо така необхідність передбачається).

Останньою загальною функцією є «Загальний збір даних». За необхідності в цій функції описується які саме дані будуть збиратися на вершині-ініціаторі після виконання обчислень, та, за потреби, описуються варіації механізму збору цих даних (наприклад операції редукування). У разі наявності «Загальний збір даних» автоматично буде викликатися після закінчення роботи потокової та акселераторної функції в динамічному об’єкті. У відповідних динамічних об’єктах створюються ці функції. Після виконання цього кроку система готова до початку обчислень.

2.6.2. Динамічне балансування

Динамічне балансування дозволяє:

1. Безпосередньо під час роботи компенсувати той відсоток неефективності знайденого статичним балансувальником рішення задачі розподілу навантаження в системі;
2. Значно збільшити стресостійкість системи, тим самим зменшити втрати в разі ситуації обриву зв’язку навіть з усіма вузлами;
3. Втілити оглянуту в першому розділі перспективну технологію відкладених обчислень в форматі значно простіше, ніж це можна було зробити на етапі статичного планування.

Також динамічний балансувальник відповідає за початок і проведення обчислень. Свою роботу він веде на кінцевих вузлах, тобто, згідно зі схемою на рисунку 2.5, на рівні паралельної системи. Проте присутній механізм централізації, при якому в деяких режимах роботи передбачається можливість часткового переходу на більш високий рівень абстракції (тобто на рівень розподіленої системи).

На першому своєму кроці динамічний балансувальник кожного вузла в рамках технології відкладених обчислень відправляє вузлу-ініціатору обчислень (безпосередньо або через проміжні вузли) запит про отримання відповідного вузлу-автору запиту динамічного об’єкту із набору динамічних

об’єктів описаного вище. Відповідно ці запити приходять на вузол-ініціатор і зберігаються в форматі асинхронної черги коллбеків. Оскільки дані, які передаються вузлам, вже розподілені так, аби врівноважити об’єм пакету даних, призначеного для конкретного вузла, з пропускною здатністю каналу зв’язку з цим вузлом, то не важливо, в якому порядку будуть видаватися дані. В програмі передбачається, що як тільки динамічний об’єкт із списку буде повністю готовий до того, щоб бути відправленим на відповідний йому вузол, то з асинхронної черги запитів буде вибрано відповідний запит, і в форматі коллбеку буде передано сереалізований динамічний об’єкт на цільовий вузол (безпосередньо або через проміжні вузли, для чого на них передбачені функції подальшого перекидання запитів).

При прийомі кінцевим вузлом призначеного йому динамічного об’єкту, цей вузол здійснює його десереалізацію та подальший поділ описаної в цьому об’єкті задачі на ще менші підзадачі. Механізм цього поділу в загальному відповідає механізму поділу задачі на етапі статичного балансування (тобто, можна сказати, що статичне балансування також в певній мірі відбувається і на рівні паралельної системи), але при цьому проміжки даних рівні між собою. Поділ відбувається на певну кількість частин, яка завідома більша, ніж кількість ядер/потоків в центральному процесорному елементі цього вузла. Всі отримані в результаті цього поділу нові динамічні об’єкти додаються в паралельну чергу виконання. Перші P об'єктів, де P – кількість ядер/потоків в центральному процесорному елементів одразу відправляються на обробку, яка полягає у виклику потокової функції, описаної в відповідному динамічному об’єкті.

Зазначимо, що у разі, якщо вузол містить акселератор, то від ініціатора він прийме два динамічних об’єкти, один з яких відповідатиме за обчислення на центральному процесорному елементі, а інший за обчислення на акселераторі. Механізм обробки об’єкту, призначеного для центрального процесорного елементу описаний вище. Механізм обробки об’єкту, призначеного для акселератору полягає в поділі за аналогічною схемою, але лише на три об’єкти.

За схемою, аналогічній представленій в попередніх дослідженнях [14-15], в системі виділяються два потоки, які будуть відповідати за співпрацю з акселератором. Необхідність цього пояснюється тим, що передбачається використання перш за все інтерфейсу PCI Express, який працює в послідовному повнодуплексному режимі, тобто два потоки необхідні для ситуації, коли один потік буде вести відправку своїх даних, а інший в той же час вже прийматиме свої результати. Звісно виникнення такої ситуації малоймовірне.

Також на кожному вузлі виділяється окремий потік-монітор, який відповідатиме за виконання операцій моніторингу стану паралельної системи та її складових. З певною завчасно обраною фіксованою частотою (наприклад раз в 10 секунд) цей потік буде виходити з блокування та проводити огляд показників елементів системи (центральних процесорів та акселераторів), таких як різниця системного показника завантаженості елемента між двома останніми замірами, різниця тактової частоти між двома останніми замірами (оскільки передбачається, що за рішенням планувальника операційної системи елемент може увійти в режим Turbo лише на певному етапі обчислень), різниця температури елементу між двома останніми замірами, стан черг обробки потокових та акселераторних функцій. Ці дані монітор передає динамічному планувальнику та відповідний потік-монітор блокується до наступного заміру.

Динамічний планувальник, відповідно до отриманих від монітору даних, приймає ряд рішень щодо подальшої обробки:

1. Якщо не досягнуто заданий в файлі конфігурації вузла показник бажаної завантаженості елементу, то на виконання відправляється, в залежності від різниці поточної і бажаної завантаженості, один або декілька динамічних об’єктів із відповідної черги елементу, або, за наявності заблокованих об’єктів, які вже оброблялись, то в першу чергу поновлюється їх обробка. У разі, якщо черга пуста, то вузол сигналізує про готовність прийняти додаткові обчислення (механізм цього описано нижче).
2. Якщо перевищено заданий в файлі конфігурації вузла показник бажаної завантаженості елементу, то, в залежності від різниці поточної і бажаної

завантаженості, блокується виконання одного або декількох динамічних об’єктів із тих, які наразі проходять обробку. Виконується ребалансування між вузлами (механізм якого описане нижче).

3. При оптимальних показниках завантаженості ведеться перевірка різниці частот. Якщо частоти збільшились, то виконуються дії, аналогічні діям, описаним в кроці 1. Якщо частоти зменшились, то виконуються дії, аналогічні діям, описаним в кроці 2.
4. Якщо частоти не змінились, то ведеться оцінка різниці температур. Якщо температура зменшилась як мінімум на 10 градусів і тепер сягає менше 60 градусів, то виконуються дії, аналогічні діям, описаним в кроці 1. Якщо температура збільшилась як мінімум на 10 градусів і тепер перевищує 60 градусів, то виконуються дії, аналогічні діям, описаним в кроці 2. Варто зазначити, що показники температури орієнтовні, і мають виставлятись згідно з попередніми спостереженнями.
5. Якщо зміна температури не перевищувала 10 градусів, то ніяких змін не відбувається, робота вузла та його елементів продовжується в стандартному режимі.
6. Монітор може запускатися також в позачерговому режимі при отриманні запиту уточнення показників (описано нижче).

Динамічне баласування виконується також частково (в саму останню чергу, тобто вищим пріоритетом володіють ті підзадачі, які вже наявні на вузлі) і між вузлами, тобто на рівні розподіленої системи. Для цього передбачається вищезгаданий механізм ребалансування. Полягає він в делегуванні обчислень одним вузлом-робітником іншому.

Сама по собі задача передачі процесів в рамках розподіленої системи є надзвичайно складною та досі повноцінно не вирішена в наш час. Це пояснюється тим, що разом із процесом необхідно передати весь його контекст, стан стеку, даних, співставити адресний простір, при цьому врахувати ще й не можливу неоднорідність архітектур вузлів. Це характерно як для процесорних елементів, так, в ще більшій мірі і для акселераторів. Тому пропонується

механізм, який полягає не в передачі частини процесу, який вже виконується, а передачі частини обчислень, які чекають в черзі на виконання.

В описаному вище алгоритмі роботи потоку-монітору вузла передбачено випадок, коли вузол переходить в режим готовності прийняти додаткові обчислення. Оскільки в системі передбачається часткова зв’язність, а не повна, то сигнал про цю готовність вузол надсилає вузлу-ініціатору, на якому ці сигнали збираються в єдиний список, який сортується за продуктивністю вершин (тобто вагою вузлів графу системи), від найпотужнішого до найслабшого. Також вузлу-ініціатору надсилаються і сигнали про перевантаженість вузла, тобто прохання від вузлів на те, щоб частину їх обчислень прийняли інші вузли. При надходженні запиту на делегування обчислень іншому вузлу, вершина-ініціатор перевіряє стан списку вузлів, які готові прийняти обчислення, та, у разі наявності у списку вузлів, обирає перший (найпотужніший) та надсилає йому уточнювальний сигнал, який запускає позачергово монітор даного вузла на перевірку поточних показників (аби уточнити, чи не застарів сигнал про готовність, тобто чи не змінились показники завантаженості вузла від часу отримання останнього сигналу від нього). У разі, якщо перевірка підтвердила готовність вузла прийняти обчислення, то запускається механізм делегування обчислень між вузлами, який буде описано нижче. У разі, якщо перевірка не підтвердила готовність вузла прийняти обчислення, то сигнал від цього вузла видаляється зі списку сигналів та перевіряється наступний за списком вузол. Додатково потім за аналогією можна проводити уточнення актуальності делегування і в вузла, від якого надходить запит.

2.6.3. Механізм делегування обчислень між вузлами

Для делегування обчислень призначені, відповідно до опису основних функцій для розпаралелювання програми, приведеного в підрозділі 2.3, останні три із описаних функцій, а саме «Часткове розділення даних», «Частковий збір

даних», «Часткове розділення задачі». Логіка їх роботи аналогічна логікі роботи відповідних функцій на рівні статичного планувальника, але розбиття відбувається не на багато проміжків даних, а лише на два рівномірних, відповідно породжуються два нових динамічних об’єкти. Один із них лишається в черзі вузла, на якому відбувається розподілення, а інший відправляється вузлу-ініціатору, який передає його вузлу, готовому прийняти обчислення, на якому при отриманні даний об’єкт одразу запускається на обробку.

Даний механізм працює як з об’єктами, призначеними для центральних процесорних елементів, так і з об’єктами, призначеними для акселераторів.

Функція збору даних в цьому випадку передбачає так само асинхронний збір даних за запитом, через механізм коллбеків. Якщо при обробці об’єктів, призначених від самого початку для вузла, збір даних ведеться спершу на цьому вузлі, а потім вже передається вузлу-ініціатору, то при обробці делегованих об’єктів результати обчислень передаються по тому ж маршруту назад до вузла, на якому планувалась обробка цього об’єкту спершу. При цьому запит на отримання даних відправляється початковим вузлом від самого початку, тобто результати будуть передані як тільки будуть готові.

Відмітимо також, що виконання описаного механізму динамічного балансування за такою схемою також забезпечує описану в першому розділі автоматичну регуляцію зерна паралелізму. Водночас з цим, нема ніяких перешкод користувачу в рамках коду, виконання якого він бажає розподілити між вузлами системи, реалізувати також паралелізм. Більше того, така реалізація може бути доцільною у випадку застосування дрібнозернистого паралелізму в форматі розпаралелювання циклів, про що свідчать дослідження [15-16].

2.7. Стресостійкість та безпека системи

Основна стресова ситуація, яка може виникнути в такій розподіленій системі це обрив зв'язку з одним із вузлів під час безпосереднього виконання обчислень. Для мінімізації загальних втрат внаслідок цього застосовується асинхронна відкладена передача результатів та прискорений моніторинг доступності вузлів.

2.7.1. Асинхронна відкладена передач результатів

Згідно з технологією відкладених обчислень, а конкретно в даному випадку її реалізації у вигляді відкладених асинхронних передач, функції часткового та загального збору даних працюють, за допомогою механізму коллбеків, за такою логікою, яка передбачає одразу по делегуванню обчислень від візла-ініціатора до кінцевих вузлів формування запитів до цих вузлів на отримання всіх часткових результатів. По закінченню кожного окремого обчислення (тобто обробки конкретного динамічного об’єкту, див. вище) не відбувається очікування закінчення інших обчислень на вузлі, а відбувається одразу передача результату до коллбеку. При цьому, завдяки тому, що на етапі статичного балансування було проведено розподіл одразу на всі вузли, а не реалізовано підрозподіл по рівнях графу (тобто кожна батькіська вершина виконувала б розподіл всім своїм дочірнім вершинам), то результат передається не батьківському вузлу, а одразу прямує до вузла-ініціатора. На вузлі-ініціаторі зберігається схема делегування обчислень за всією системою, і відповідна їй схема прийому даних-результатів.

Керування чергою безпосередніх відправок при такому підході покладається на планувальник операційної системи.

Також при такому підході у випадку, коли відбувається обрив зв’язку з одним із вузлів, вузол-ініціатор перевіряє, які результати від цього вузла та дочірніх йому вузлів були прийняті, і порівнює з тим, які планувалось прийняти.

Динамічні об’єкти, результати яких були не прийняті, перерозподіляються по доступним вузлам як нова задача, для якої виконується перебудування системи на логічному рівні та статичний перерозподіл динамічних об’єктів на доступні вузли. При цьому ці об'єкти отримують нижчий пріоритет, ніж ті об’єкти, які вже є в чергах обробки динамічних об’єктів, наявних на вузлах, тобто будуть виконуватись в останню чергу, аби мінімально порушувати заплановане початкове балансування в системі.

2.7.2. Прискорений моніторинг доступності вузлів

Для поточної перевірки доступності вузлів, з метою своєчасного виявлення обривів проводиться централізоване опитування вузлом-ініціатором всіх вузлів, наявних в системі. Полягає воно в звичайній спробі встановити мінімальний зв’язок (в рамках прийнятого протоколу обміну сигналами в системі) з вузлом з певною періодичністю таких спроб (наприклад раз в 10 секунд). Якщо зв’язок з якимсь із вузлів встановити не вдається, то з мінімальною затримкою (1-2 секунди) відбуваються повторні спроби. Після декількох невдалих спроб зв’язок з вузлом вважається обірваним.

2.7.3. Безпека системи

Як зазначалося у вступі та поставленому завданні, розроблений метод повинен бути націлений в тому числі на реалізацію в розподілених комп’ютерних системах, вузлами яких можуть являтися звичайні персональні комп’ютери сторонніх людей, які виявили бажання надати потужності своїх машин для певних ємкісних обчислень, і самі при цьому можуть ініціювати обчислення. Тому важливим питанням є безпеки, оскільки немає гарантій, що в код, який буде виконуватись на всіх машинах, не буде закладено шкідливого або шпигунського функціоналу.

Для уникнення цієї вразливості можна запропонувати ввести на етапі аналізу абстрактного синтаксичного дерева перевірку залежностей паралельного коду, виконання якого передбачається на вузлах. Перевірка полягатиме в обмеженні паралельним функціям доступу до використання сторонніх бібліотек, а також вбудованих в конкретну мову програмування, на якій відбувається реалізація описаного методу просторів імен, пакетів, бібліотек, тощо, які дозволяють програмному коду взаємодіяти безпосередньо із системою.

Оскільки передбачається робота системи за схемою розподілення обчислень паралельних секцій (рис. 2.1), то таке обмеження створює умову того, що користувацький код повинен розподіляти для паралельного виконання в системі лише обчислення, не маючи змоги створювати файли, генерувати новий код, впливати на систему безпосередньо.

Але при цьому обмеження повинні стосуватися лише прикладного коду, а не системного, оскільки на системному рівні програми взаємодія із системою вузла природньо необхідна.

Альтернативою може бути контейнеризація всього програмного коду, аби він мав доступ лише до контейнеру, а не системи безпосередньо, але це рішення пов’язано з рядом труднощів, оскільки на рівні системного програмного забезпечення все ж повинен бути доступ до безпосередніх параметрів системи та акселераторів. Також контейнеризація потенційно може виказати значний негативний вплив на швидкодію розподіленої системи в цілому.

Також в системі передбачається стандартний інтерфейс функцій, які породжують безпосередні контакти (обмін пакетами) між вершинами, та загальний формат можливих запитів та відповідей. Тому будь-які сторонні та нестандартні запити будуть відхилятись, а отже пакетна безпека буде покладена на мережеві засоби безпеки на рівні операційних систем та файрволів окремих вузлів.

Висновки до розділу 2

В даному розділі було описано метод організації структури та режиму роботи з довільними обчисленнями розподіленої гетерогенної комп’ютерної системи. Складовими вузлами такої системи є довільні паралельні обчислювальні системи, які в тому числі можуть містити акселератори. Даний метод нерозривно пов’язаний з методом балансування навантаження на різних абстрактних рівнях системи, що є ключовим аспектом ефективного функціонування гетерогенної розподіленої системи.

На відміну від розглянутих у першому розділі сучасних засобів та технологій організації паралельних обчислень в розподілених комп’ютерних системах, які всі орієнтовані на гомогенність системи та не підтримують в повній мірі гетерогенність, в запропонованому підході підтримується як гетерогенність на рівні центральних процесорних елементів, так і гетерогенність в плані наявності різнорідних обчислювачів (тобто акселераторів), і, відповідно, враховується також гетерогенність таких акселераторів. Крім того, також на відміну від розглянутих технологій, від топології системи не вимагається повнозв’язності, система може мати довільну топологію, яка крім того, лишаючись незмінною на фізичному рівні, має змогу перебудовуватись на логічному рівні для приведення до єдиної деревовидної структури, коренем якої є вузол-ініціатор обчислення. Це дозволяє забезпечити рівноправність вузлів системи, тобто кожен вузол системи може виступити ініціатором обчислень в ній.

Виділення додаткового рівня абстракції в такій системі дозволило проводити ефективне планування навантаження на різних її рівнях. При цьому для забезпеченості більш ефективного вирішення задачі планування проводиться і статичне і динамічне планування, які виконуються як в рамках одного рівня, так і в рамках декількох рівнів. В рамках динамічного балансування реалізовано також механізм автоматичної регуляції зерна паралелізму на конкретних вузлах.

Для ефективної роботи планувальників запропоновано комплексне оцінювання як вхідної задачі, так і всіх елементів системи, при цьому оцінювання елементів враховує особливості задачі, розв’язання якої планується в системі.

Оскільки механізм міграції процесів не тільки в розподіленій гетерогенній системі, а й навіть в гомогенній являється надзвичайно складною задачею, то запропонований компромісний варіант ребалансування обчислень між різними вузлами системи – за допомогою розподілення обчислень та перепризначення зі зберіганням структури розподілу.

Передача даних в системі реалізована в контексті технології відкладених обчислень, а конкретно у варіанті відкладених асинхронних передач, з централізацією їх збору та механізмом перекидання передач між вузлами. Планування таких передач та їх планування покладено на планувальник нижчого рівня, тобто рівня операційних систем вузлів.

Для підвищення стресостійкості системи у випадках втрати зв’язку з одним вузлом або групою вузлів запропоновано підхід до уникнення аварійної зупинки системи та мінімізації втрат внаслідок розриву, шляхом ребалансування невиконаних обчислень на вузли, які лишились доступні, з попередньою оптимізацією розподілу.

Для підвищення захищеності вузлів системи пропонується на етапі аналізу прикладного програмного коду, паралельне виконання якого передбачається в системі, відхиляти використання в коді додаткових засобів (бібліотек, пакетів, модулів, тощо) конкретних мов, які дозволяють прикладному коду безпосередньо впливати на систему.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

3.1. Розробка архітектури прототипів

В описі запропонованого методу, який наведено в попередньому розділі, вже створено заділ для загальної схеми архітектури прототипів.

Система повинна являти собою рівноправну систему, тобто програмне забезпечення кожного вузла повинно мати змогу працювати як в режимі серверу (як вузол-ініціатор обчислень), так і в режимі клієнту (як вузол-приймач обчислень). В цілому достатньою умовою є наявність як мінімум одного вузлу із серверним забезпеченням, але тоді ініціатором обчислень в системі зможе виступати лише цей вузол.

Відповідно до аналізу, проведеного в першому розділі, можна відмітити один з суттєвих недоліків архітектури сучасних розподілених комп'ютерних систем – сервіс-орієнтованість їх програмного забезпечення. Тобто, виділяється повністю окрема програма, яка забезпечує обчислення, окрема програма яка відповідає за керування цими обчисленнями, балансування навантаження та планування обчислень, і окрема програма-ініціатор. Основним недоліком такого підходу є необхідність встановлення всіх цих окремих програмних комплексів та реалізації зв'язку між ними користувачем вручну, що може породжувати помилки та невідповідності.

Пропонується зберегти сервісно-компоненту архітектуру всередині програмного комплексу, навіть з можливість запуску всіх компонентів в окремих процесах. Але з точки зору користувача надавати все це як єдиний повнозв'язний програмний продукт.

Компонентами, згідно з попереднім розділом, являються наступні програмні елементи:

1. Ініціалізатор системи;
2. Окремий підкомпонент ініціалізатора – тестувальник потужності;

3. Аналізатор прикладного коду;
4. Балансувальник навантаження;
5. Окремий підкомопонент балансувальника – монітор стану системи.

Всі інтерфейси взаємодії між вищезазначеними компонентами повинні відповідати задекларованим в попередньому розділі зв’язкам між етапами функціонування системи.

3.2. Обґрунтування вибору засобів реалізації прототипів

Відповідно до аналізу, проведеного в першому розділі, можна відмітити, що основними засобами програмування, які використовуються при розробці програмного забезпечення для розподілених комп’ютерних систем є MPI (або на рівні розподіленої системи, подекуди у зв’язці з OpenMP на рівні паралельної системи, або повністю виключно MPI) та Intel TBB. Окрім зазначеного в першому розділі недоліку, пов’язаного з орієнтованістю цих засобів на гомогенні повнозв’язні системи і відсутності вбудованої підтримки акселераторів на рівні планування та балансування обчислень в системах, можна виділити ще додаткові недоліки, які випливають з цих використання цих технологій.

Не зважаючи на високу швидкодію програм, організованих з використанням цих засобів, та загальну зручність їх застосування, основною їх проблемою з точки зору програмування лишається застарілість їх орієнтації, яка з року в рік тільки посилюється. Серед ключових моментів, які вказують на це, можна виділити три, кожен наступний випливає з попереднього:

1. Обмеженість цільових мов. OpenMP та MPI перш за все розроблялись для мов Fortran та C, Intel TBB – C++. Fortran наразі повністю втратив популярність при розробці прикладного програмного забезпечення, застосування мови C перейшло в сферу системного програмного забезпечення та підтримки існуючого коду. ввтратили популярність та актуальність в розробці прикладного програмного забезпечення, про що

свідчать статистики за останні роки, зібрані такими сайтами як Google, GitHub, StackOverflow.

2. Орієнтованість OpenMP та, перш за все, MPI на процедурне програмування. Основи обидвох технологій розроблялися в часи, коли основною парадигмою, яка застосовувалась в прикладному програмуванні, була процедурна. Проте наразі загальноприйнятими вважаються значно складніші парадигми – об'єктно-орієнтовна, функціональна, реактивна. Хоча розробка продовжується і понині, але через необхідність підтримки зворотної сумісності про повноцінну підтримку сучасних підходів мова не йде.
3. Орієнтованість Intel TBB на C++. Хоч C++ і лишається потужною та високопродуктивною мовою, яка досі розвивається та переймає в себе сучасні підходи до розробки прикладного програмного забезпечення, і яка досі де-факто лишається стандартом для високонавантажених обчислень, але з року в рік нові мови програмування наздоганяють та подекуди й переганяють C++ в питаннях продуктивності, про що свідчать наприклад такі дослідження як [15-16]. В той же час C++ лишається складною мовою, яка вимагає значно вищої кваліфікації розробників, і куди менш зручніше застосування сучасних парадигм (знову ж таки через те, що на етапі закладення основ C++ застосування і підтримка таких парадигм не передбачалась, а через проблему зворотної сумісності реалізація її не завжди є найелегантнішою і найпростішою), ніж в більш новітніх мовах. Те саме можна сказати і про Intel TBB – хоч він і являється потужним інструментом для програмування паралельних систем (і при тому лише зі спільною пам'яттю), але наступний недолік (п. 4) стосується і його.
4. Низький рівень абстракції, необхідність виділення значного проміжного рівня в програмному комплексі для організації взаємодії між шаром бізнес-логіки та шаром комунікації, приведення інтерфейсів, тощо. Наприклад, в MPI максимальна абстракція, яку можна відправити – завчасно визначена

структура. Все це вимагає від розробника ручного вирішення, високої кваліфікації, та породжує великий простір для потенційних помилок.

Описаний в другому розділі роботи запропонований метод являє собою сукупність послідовних кроків та рішень для реалізації певної заданої в ТЗ мети, і при розробці був від самого початку орієнтований в тому числі на можливість реалізації засобами будь-якої сучасної високорівневої мови програмування, тобто згідно з сучасним хорошим підходом відсутності прив'язаності архітектури до реалізації. В мові є бажаною лише підтримка функціонального програмування в форматі можливості передавати функції в якості аргументів іншим функціям, а також наявність можливості створення динамічних об'єктів засобами мови реалізації. Проте обидві вимоги не є обов'язковими і в разі відсутності такого функціоналу його можна компенсувати наприклад засобами автоматичної генерації коду.

Якщо в системі передбачається наявність акселераторів, то від мови програмування прототипів вимагається наявність засобів (вбудованих або сторонніх) для організації обчислень на акселераторах. Наприклад, у випадку, якщо акселераторами передбачаються графічні процесори, то вимагається наявність реалізації таких технологій як OpenCL або CUDA у вигляді бібліотек, сумісних з обраною мовою.

В рамках даного розділу пропонується реалізація запропонованого методу засобами мови C# (конкретно технології WCF (Windows Communication Foundation) та TPL (Thread Parallel Library) в рамках фреймворку .NET Framework компанії Microsoft). В цільових тестових системах передбачається використання акселераторів у виді графічних процесорів, тому для їх підтримки використовується бібліотека OpenCL.

Даний вибір пояснюється наступними чинниками:

1. Мова C# є сучасною високорівненою мультипарадигмовою мовою програмування, в якій надається весь необхідний функціонал для реалізації запропонованого методу;

2. Серед передбаченого функціоналу методу присутня автоматична зміна зерна паралелізму (фактично – його зменшення до оптимального розміру для кожного конкретного вузла в рамках кожного окремого обчислення). Попереднє тестування, проведене в рамках досліджень [15-16] показало, що при реалізації програм із зменшеним зерном паралелізму засобами мови C# можна досягти кращої швидкодії, ніж засобами C++ та OpenMP;
3. Технологія WCF дозволяє організувати ефективну комунікацію та виконання делегованих обчислень (в тому числі відкладених) в розподілених комп'ютерних системах, окрім того дозволяючи поєднувати різнорідні елементи, як то системи на базі процесорів Intel та AMD, з різними операційними системами, тощо; При цьому вимагає організації строго вивіреного інтерфейсу взаємодії вузлів, що задовольняє питанням безпеки, оглянутим в підрозділі 2.7.3; Також в рамках технології WCF надається широкий спектр оптимізованих протоколів передачі даних між вузлами в форматі протоколів SOAP (Simple Object Access Protocol), серед яких SOAP на основі TCP (Transmission Control Protocol), SOAP на основі UDP (User Datagram Protocol), SOAP на основі HTTP (HyperText Transfer Protocol), SOAP на основі Message Queues, тощо;
4. Виконання програм, написаних засобами .NET Framework, відбувається в рамках віртуальної машини CLR (Common Language Runtime), що може дещо негативно впливати на швидкодію, але розширює спектр сумісних мов, виконує ізоляцію програми від безпосередньо системи, цим самим наприклад створюючи єдину модель пам'яті для всіх систем, зовнішню незалежність від архітектури процесорів вузлів, тощо. Якщо в розподіленій системі передбачається наявність вузлів з різними операційними системами, то можна застосувати більш розширену версію .NET Framework - .NET Core, яка орієнтована на кроссплатформність;
5. Водночас в .NET Framework є всі необхідні засоби для роботи з API операційної системи, що дає можливість проводити оцінку параметрів і стану системи, відповідно до підрозділів 2.5.2 та 2.6.2;

6. Незалежність реалізацій від версії та виробника компілятору. Наприклад для мови C++ існують компілятори GNU, компілятори від компаній Intel, Microsoft, тощо. Поведінка програм може дещо відрізнятись в залежності від того, яким компілятором її компілювали. І немає ніяких гарантій, що на всіх вузлах системи буде встановлено один і той же компілятор C++ однієї і тієї ж версії. Для мови C# доступний єдиний і офіційний компілятор від Microsoft, тому проблема може бути лише в несумісності версій, але не в загальній несумісності компіляторів;
7. В мові C# підтримуються всі розглянуті в підрозділі 1.7.4 формати відкладених обчислень, в тому числі асинхронні паралельні обчислення. В технології WCF від початку закладено реалізацію відкладених передач даних;
8. В мові C# широко підтримується застосування відвантажених обчислень, за рахунок наявності підтримки реалізації бібліотеки OpenCL;
9. В мові C# надається можливість створення динамічних об’єктів (DynamicObject та ExpandoObject стандартного пакету System.Dynamic), що спрощує розробку. З точки зору процесу програмування робота з цими об’єктами значно простіша, ніж робота з аналогічними примітивами в C++;
10. Зручне розгортання програм, реалізованих засобами C# WCF в хмарних середовищах, що дозволяє проводити значно ширше тестування з різноманітними комбінаціями складу тестових цільових розподілених систем.

3.3. Опис контрольних прототипів

Для перевірки ефективності всіх запропонованих в розділі 2 методів та підходів необхідно реалізувати їх в контексті моделювання роботи реальних програм для організації паралельних обчислень в реальних розподілених гетерогенних комп’ютерних системах із акселераторами, та в рамках цих моделей програм провести вирішення різноманітних математичних та

прикладних задач із різними вхідними параметрами, для виявлення залежностей та прозорої оцінки ефективності цих програм.

Також необхідно порівняти запропоновані підходи і методи з певними контрольними зразками, тобто програмами для організації паралельних обчислень в реальних розподілених гетерогенних комп'ютерних системах із акселераторами, в рамках яких теж проводиться вирішення відповідних різноманітних математичних та прикладних задач із різними вхідними параметрами, але вже без застосування запропонованих методі та підходів оптимізації. Різниця в показниках швидкодії, прискорення та ефективності між цими контрольними програмними прототипами і буде вказувати на реальну ефективність запропонованих підходів та методів.

Як було вказано в першому розділі, в тій же негативній мірі, в якій критичні секції впливають на швидкодію, коефіцієнти прискорення та ефективності паралельних програм, реалізованих за моделями зі спільною пам'яттю, так пересилки даних негативно впливають на відповідні показники систем із локальною пам'яттю. Плюс до того, як відмічено в другому розділі, обмін даними між вузлами через глобальну мережу не завжди може виконуватись за однакові проміжки часу. Тому, задля виявлення середніх діапазонів відхилення латентності додатково виконуються заміри часу на пересилки даних, шляхом вирахування різниці часу відправлення та часу прийому. Задля мінімізації впливу заміру часу на роботу програми виконання всіх замірів на всіх вузлах виконується в паралельних асинхронних функціях, а обмін результатами та їх співставлення відбувається вже після закінчення безпосередніх обчислень.

Відповідно, реалізовано пакет із чотирьох програмних модельних комплексів для організації паралельних обчислень в розподілених гетерогенних комп'ютерних системах із акселераторами, із застосуванням різних оглянутих в першому розділі сучасних засобів, а також із застосуванням запропонованих підходів і засобів. Характеристики цих програмних комплексів зведено в таблиці 3.1.

Таблиця 3.1.

Характеристики тестових та контрольних програмних комплексів

№ п. п.	Мова реалізації	Технологія організації паралелізму на рівні розподіленої системи	Технологія реалізації паралелізму на рівні паралельних систем	Технологія реалізації акселераторних обчислень	Підхід до балансування навантаження	Призначення
1	C#	WCF	TPL	OpenCL	Запропонований метод	Тестовий
2					Лише засобами застосованих технологій та операційних систем вузлів	Контрольний
3	C++	MPI	OpenMP			
4			Intel TBB	Intel TBB		

3.4. Опис тестової системи

Тестові системи розгорнуто засобами хмарного середовища Google Cloud Console. В рамках цього середовища надається можливість розгортати екземпляри віртуальних машин, в тому числі із акселераторами (графічними процесорами) в різних точках Землі, що є доцільним для тестування, оскільки в роботі розглядаються якраз ті загальні розподілені системи, в яких передбачається можливість будь-яких комп’ютерів у будь-якій частині світу стати вузлом такої системи.

Параметри створених розподілених гетерогенних комп’ютерних систем зведено до таблиці 3.2. Карту розміщення та принципових (заданих в рамках моделі) зв'язків між вузлами представлено на рисунку 3.1.

Таблиця 3.2

Параметри цільових тестових систем

Система 1 (тестова)				
Вузол	**Розміщення**	**CPU**	**GPU**	**ОЗП**
1	Лос-Анджелес, США	4-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla K80	15 Gb
2	Бельгія	8-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla P100	30 Gb
3	Гонконг, КНР	16-ядерний віртуальний процесор архітектури Intel SkyLake	2 шт. Nvidia Tesla K80	60 Gb
4	Сідней, Австралія	32-ядерний віртуальний процесор архітектури Intel SkyLake	-	120 Gb
Система 2 (контрольна)				
Вузол	**Розміщення**	**CPU**	**GPU**	**ОЗП**
1	Лос-Анджелес, США	8-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla K80	30 Gb
2	Бельгія	8-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla K80	30 Gb
3	Гонконг, КНР	8-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla K80	30 Gb
4	Сідней, Австралія	8-ядерний віртуальний процесор архітектури Intel SkyLake	Nvidia Tesla K80	30 Gb

Рис. 3.1. Карта розміщення вузлів тестових систем з відображенням умовних логістичних зв’язків між їх вузлами

На всіх вузлах створених розподілених гетерогенних комп'ютерних систем встановлено наступне ідентичне програмне забезпечення:

1. Операційна система: Windows Server 2012 R2 Datacenter x64;
2. .NET Framework 4.7;
3. Компілятор C++ Microsoft Visual C++;
4. OpenCL 2.2;
5. OpenMP 3.0;
6. Microsoft MPI 9.0.1.

Також зауважимо, що виходячи із топології на рисунку 3.1, в усіх тестових випадках вузлом-ініціатором обчислень будемо вважати вузол 2 (Бельгія), при цьому врахуємо, що він не має безпосередньої можливості передавати дані до вузла 4 (Сідней), і тому вся комунікація між вузлами 2 та 4 відбуватиметься через вузол 3 (Гонконг).

3.5. Опис процесу тестування

Процес тестування полягає в багаторазовому почерговому запуску всіх модельних програмних прототипів із замірами їх часу виконання, та визначенні середнього часу виконання кожної програми як середнього арифметичного від усіх часових результатів її роботи.

Також, відповідно до описаного вище підходу коротких асинхронних паралельних замірів часу початку та кінця пересилок вівся збір часових показників тривалості пересилок, для визначення їх впливу. Також вівся контроль за результуючими середніми показниками завантаження акселератору (графічного процесору) кожного із вузлів задачами, задля вирахування того, який обсяг обчислень система визначає як доцільний до виконання на акселераторі.

Часові показники результатів прямої швидкодії програми для зручності аналізу та більш прозорого відображення зведено до показників абсолютного коефіцієнту прискорення паралельної програми. Він являє собою відношення

часу виконання послідовного варіанту програми до часу виконання паралельної програми на P обчислювачах, і показує у скільки разів скорочується час виконання програми в паралельній системі [17, 30]. Розраховується коефіцієнт прискорення k_{SU} за формулою (10):

$$k_{su} = \frac{T_1}{T_P}\ , \tag{10}$$

де T_1 – час роботи послідовного варіанту програми, T_P – час роботи паралелього варіанту програми на P обчислювачах (потоках, вузлах, тощо).

В процесі тестування велася перевірка залежності показників коефіцієнтів прискорення різних програм при виконанні різних задач, в залежності від об'ємності цих задач, яка визначалася певними вхідними даними чи умовами.

Зауважимо, що при дуже великих обсягах послідовних варіантів задач, у зв'язку з неможливістю обрахувати таку велику послідовну задачу за адекватний час, час роботи послідовного варіанту обраховувався як передбачення значення функції на основі логістичного порівняння.

Іншим важливим показником ефективності роботи паралельних систем та алгоритмів є коефіцієнт ефективності k_{EF} [17, 30], який вираховується за формулою (11) на основі коефіцієнту прискорення, та показує сумарну ефективну загрузку всієї системи.

$$k_{EF} = \frac{T_1}{T_P * P} * 100\%\ , \tag{11}$$

де T_1 – час роботи послідовного варіанту програми, T_P – час роботи паралелього варіанту програми на P обчислювачах (потоках, вузлах, тощо).

Проте, оцінка коефіцієнту ефективності для паралельних розподілених гетерогенних комп'ютерних систем із акселераторами не може дати достатньо прозорого відображення, оскільки такі потужні обчислювачі, як графічні процесори, виступаючи в ролі акселераторів, будуть за формулою враховуватись в тій же самій мірі, що і звичайне окреме ядро чи потік центрального процесору.

Тому пропонується скористатись оцінкою непрямих показників, таких як коефіцієнт завантаженості акселератору (який являє собою відсоток обчислень,

які були виконані всіма акселераторами системи відносно загального обчислення), а також коефіцієнтом використання мережі (який являє собою відсоток сумарного часу, витраченого під час розв’язку задачі системою на обмін даними між вузлами та елементами від сумарного часу вирішення цієї задачі).

Окремо для розподіленої системи також необхідно обрахувати розкид латентності, тобто відсоткові показники відхилень тривалості передач одних і тих же даних між одними і тими ж обчислювачами. Це доцільно зробити розставивши часові заміри у відповідних місцях програми, і оскільки на першому передбачається багаторазовий запуск програми з одними і тими ж вхідними даними, деякими з цих етапів і можна скористатись, вважаючи їх аналогічними, відповідно і оцінювати показники часових затрат на пересилки даних на цих етапах.

3.6. Опис тестових задач та проведення експериментів

Тестування пропонується провести на шістьох різних поширених прикладних задачах, кожна з яких має свою цінність для тестування, оскільки відповідає тому чи іншому поширеному випадку в практичній організації паралельних програм в розподілених системах.

3.6.1. Векторно-матричні операції

Знаходження добутку двох квадратних матриць

Знаходження добутку двох квадратних матриць є класичною задачею для паралельного програмування, яка добре піддається розпаралелюванню.

Добуток MC двох квадртаних матриць MA та MB розмірностями N*N елементів складається з усіх можливих комбінацій скалярних добутків рядків матриці MA і стовпців матриці MB. Елемент матриці MC з індексами i, j є скалярним добутком i-го рядка матриці A і j-го стовпця матриці B.

Виходячи з цього, можна побудувати паралельний алгоритм множення двох матриць за принципом поділу одної із них за стовпцями чи рядками на частини. Тобто:

1. Всім підзадачам надсилається в повному обсязі матриця MA, та певний послідовний проміжок рядків матриці MB від i*h до 2*i*h, де i – порядковий індекс підзадачі, h = ⌈N/P⌉, а P – кількість підзадач. Тобто кожна підзадача прийматиме прямокутну матрицю MBh, розмірністю ⌈N/P⌉ рядків на N стовпців.
2. Кожна підзадача обраховує добуток прямокутної матриці MCh=MA*MBh. Тобто MCh являє собою i-тий проміжок загального результату MC.
3. Кожна підзадача відправляє обчислений проміжний результат ініціатору обчислень (вузлу або головному потоку);
4. Ініціатор обчислень встановлює елементи кожного прийнятого від підзадач проміжного результату MCh на відповідне місце остаточного результату MC.

Необхідність даного алгоритму для тестування пояснюється тим, що він дозволяє здійснити розподіл даних між вершинами не тільки фіксованого розміру частинок, а й довільного, дозволяє завантажити обчислювачі, оскільки в процесі обчислень не вимагає жодних проміжних передач даних, що дозволяє максимально прозоро обрахувати коефіцієнти прискорення та ефективності. Також він є легким в передбаченні кількості мінімально необхідних елементарних арифметичних операцій, яка для послідовного варіанту становить N^3, де N – розмірність матриць, а для паралельного для кожної окремої підзадачі – N^2*⌈N/P⌉, де P – кількість підзадач.

Відносною одиницею навантаження на підзадачу для цього алгоритму є частина рядків матриці.

В контексті описаного в розділі 2 алгоритму кожен вузол та елемент отримають такий проміжок рядків матриці MB, який відповідатиме відносному показнику продуктивності цього елементу.

На рисунку 3.2 зображено графік залежності коефіцієнту прискорення даної програми від значення розмірності матриць (N) для кожного програмного комплексу, в якому велося розв’язання.

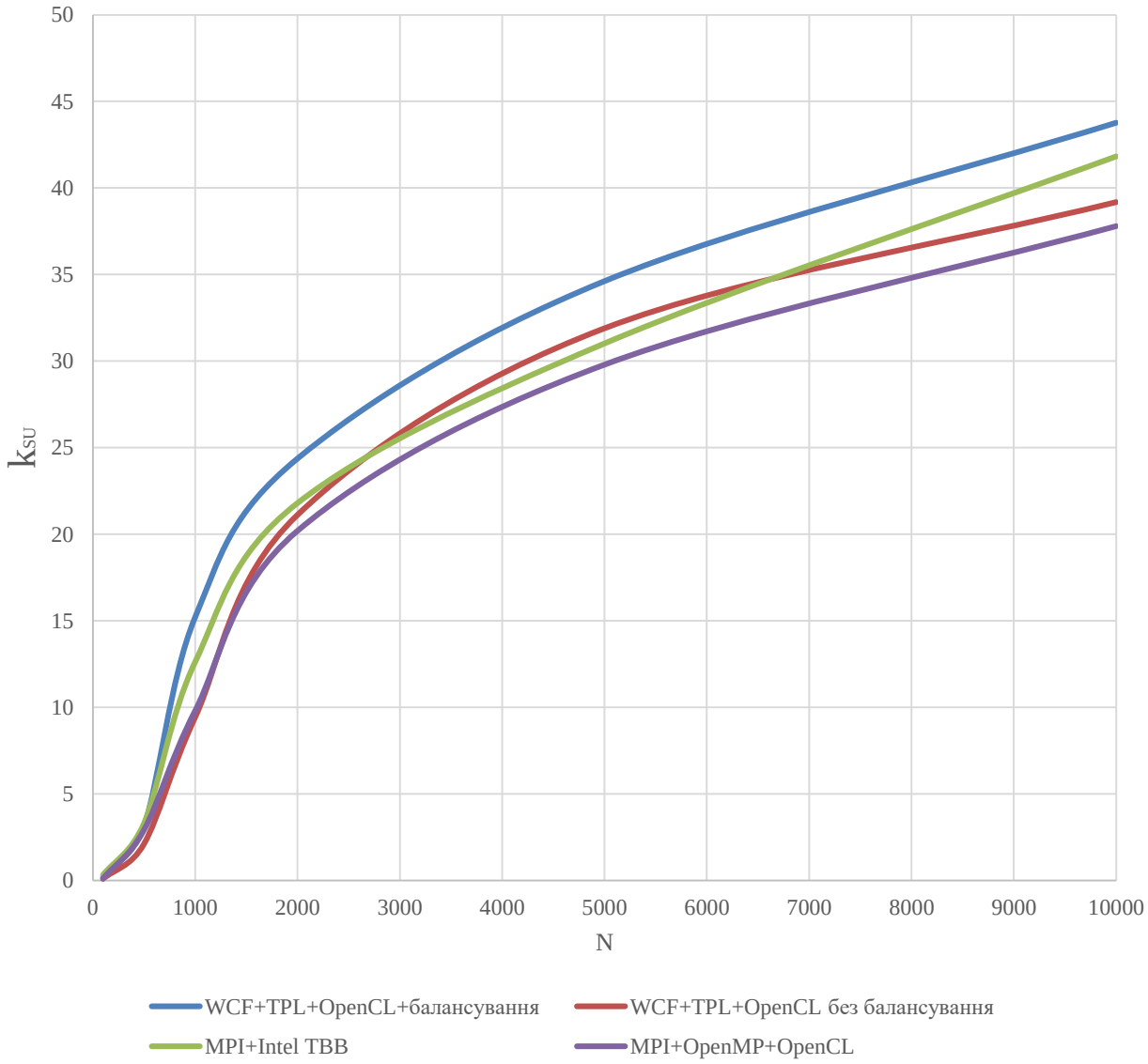

Рис. 3.2. Графік залежності коефіцієнту прискорення програми множення матриць від значення розмірності матриць (N) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.3 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення розмірності матриць (N) для кожного програмного комплексу, в якому велося розв’язання.

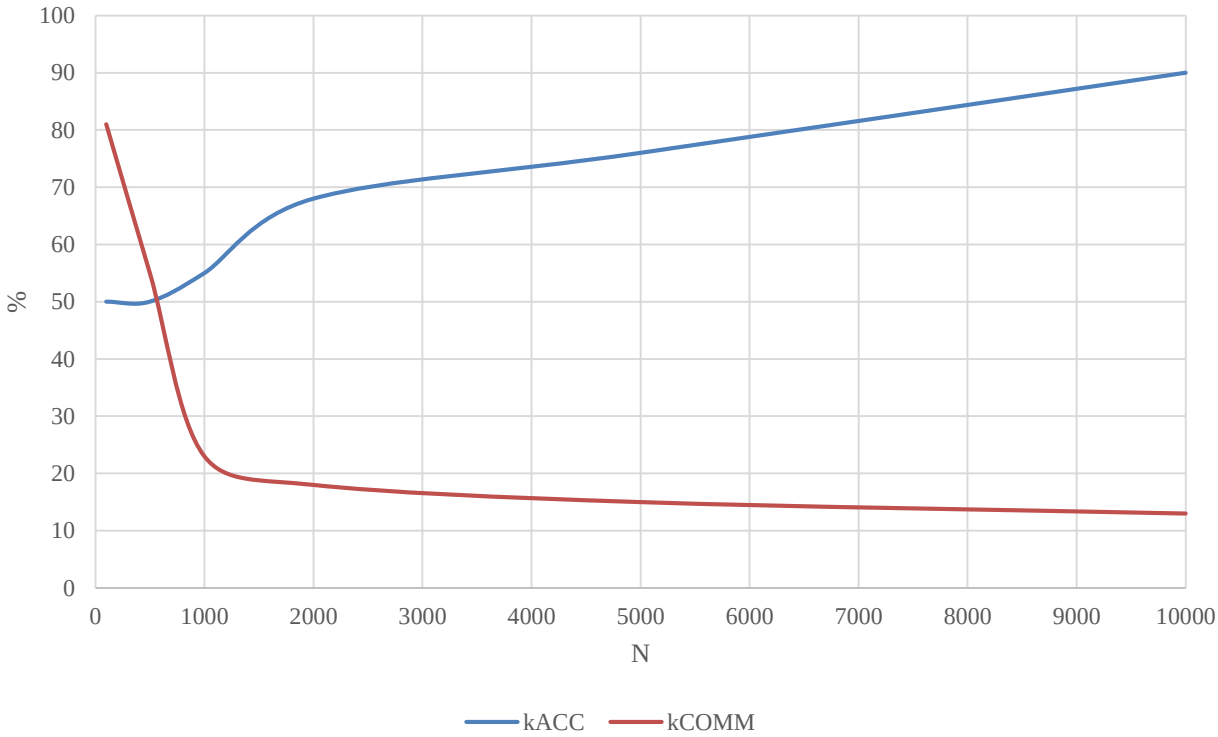

Рис. 3.3. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми множення матриць від значення розмірності матриць (N) для кожного програмного комплексу, в якому велося розв’язання

Проведення векторно-матричних обчислень за заданою формулою

Обчислення векторно-матричних формул також є розповсюдженою задачею для паралельного програмування. Її необхідність для цього тестування диктується тим, що при обчисленні такої формули присутні комунікації між обчислювачами.

Наприклад візьмемо наступний вираз C = sum(V)*MA*R+max(K)*MB*D, де MA, MB – квадратні матриці розмірністю N*N, R,D,V,K – вектори розмірністю N, sum – операція знаходження суми всіх елементів вектору, max – операція знаходження максимального елементу вектору, C – вектор-результат.

Результатом множення матриці розмірністю N*N на вектор розмірністю N є вектор розмірністю N. За аналогією з попереднім підрозділом, для паралельного виконання цієї операції необхідно розбити один із операндів, аби не передавати його повністю. В даному випадку розбиватись буде матриця-операнд по рядкам, а в повному обсязі копіюватись всім підзадачам буде

вектор-операнд. На наступному кроці можна домножити всі елементи вектору-результату на скаляр, який отримається в результаті виконання функцій sum і max.

Але для більш оптимального виконання паралельного алгоритму спершу необхідно обрахувати в паралельному режимі значення sum(V) і max(K). Для цього ініціатор обчислень (головний потік або вузол) має розіслати всім підзадачам частини (які згідно з запропонованим методом визначаються відповідно до потужності вузлів) векторів V і K. Кожен елемент системи знайде проміжне максимальне значення своєї частини вектору K та проведе підсумовування своєї частини вектору V. Ці часткові результати будуть передані ініціатору обчислень, який співставить їх та проведе остаточну відповідно операцію підсумовування чи знаходження максимуму часткових результатів.

Відносними одиницями навантаження на підзадачу для цього алгоритму є частина рядків матриці та частина елементів вектору.

На рисунку 3.4 зображено графік залежності коефіцієнту прискорення даної програмивід значення розмірностей векторів матриць (N) для кожного програмного комплексу, в якому велося розв’язання.

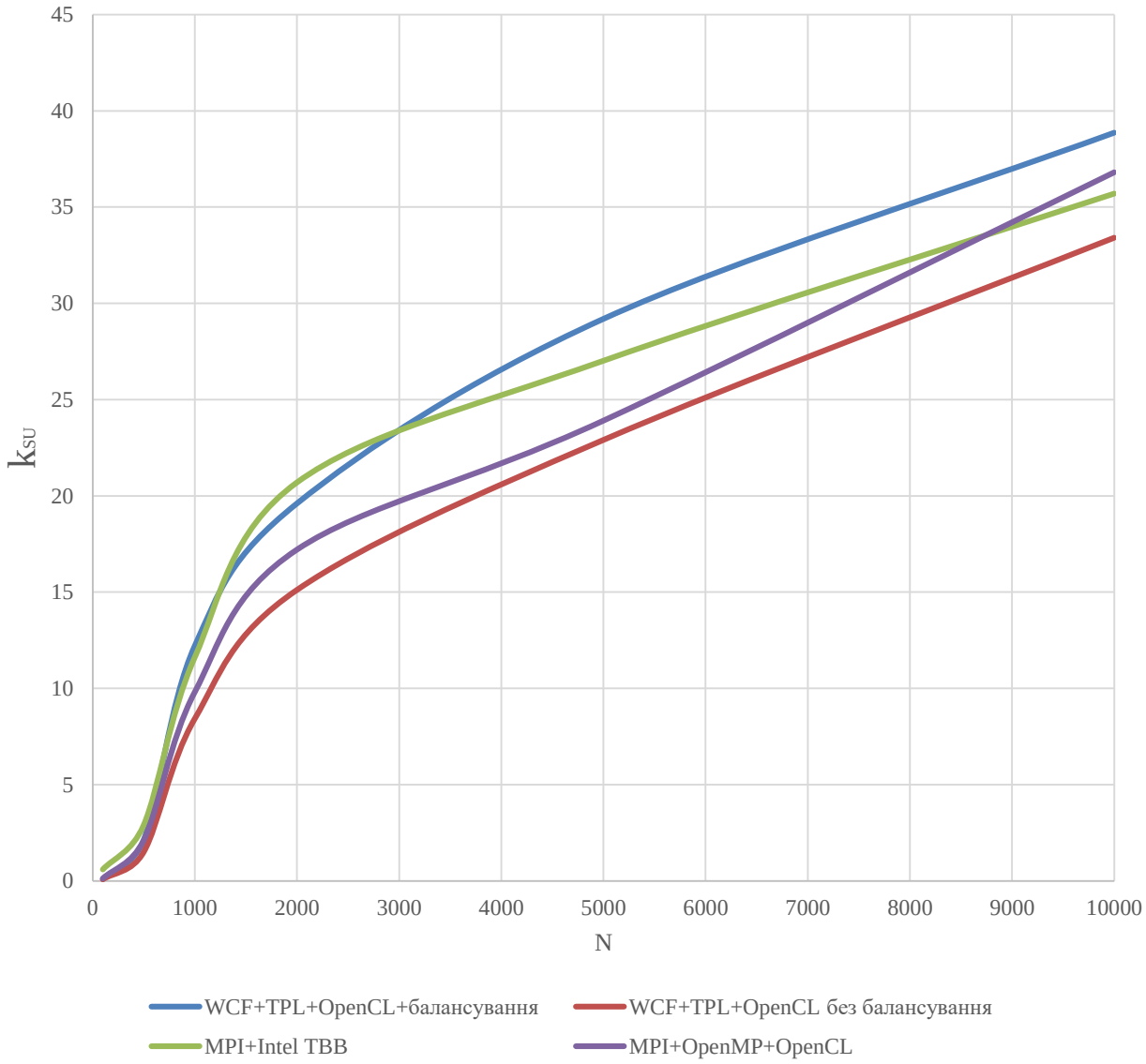

Рис. 3.4. Графік залежності коефіцієнту прискорення програми обрахування векторно-матричної формули від значення розмірностей векторів та матриць (N) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.5 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення розмірностей векторів та матриць (N) для кожного програмного комплексу, в якому велося розв’язання.

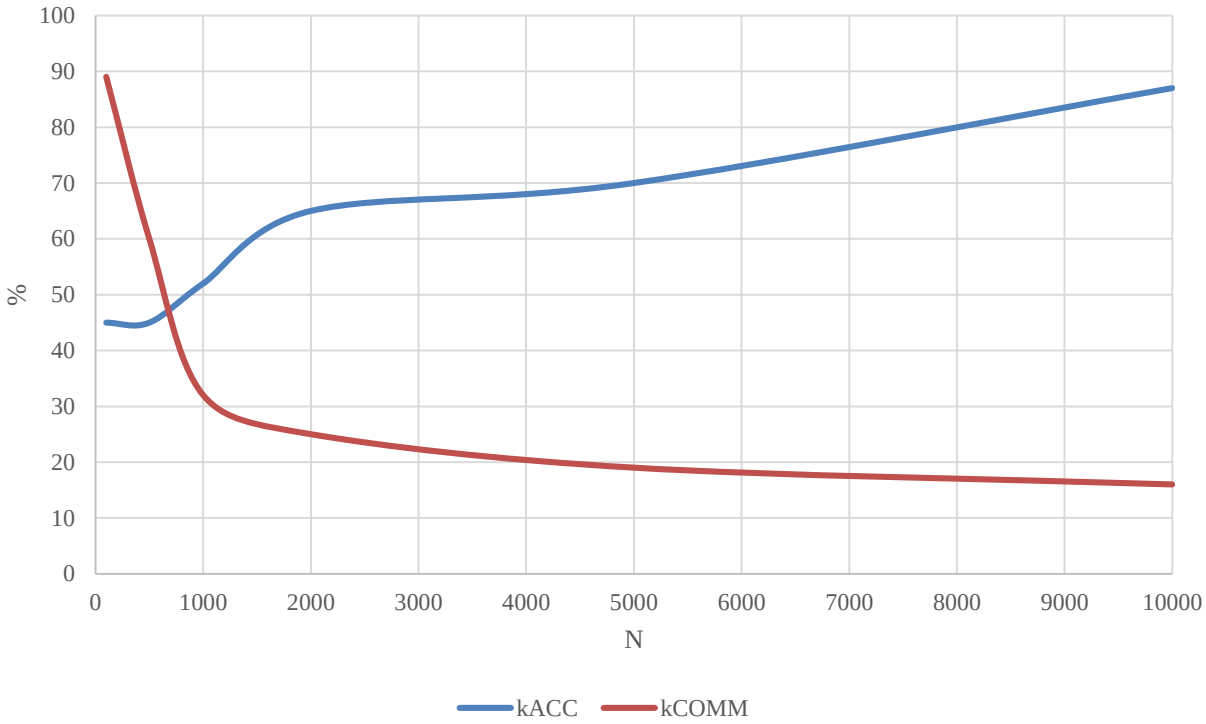

Рис. 3.5. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми обрахування векторно-матричної формули від значення розмірностей векторів та матриць (N) для кожного програмного комплексу, в якому велося розв'язання

3.6.2. Обчислення наближення ряду

Обчислення значень деяких математичних та тригонометричних функцій в обчислювальній техніці часто не можливе в прямому виді. Проте розклад в ряд та обчислення наближення цього ряду дозволяє обчислювальним машинам проводити вирахування значень математичних та тригонометричних функцій із заданою точністю.

Для прикладу візьмемо формулу обчислення гіперболічного арктангенсу для значень на проміжку від -1 до 1. Обчислення проводитимемо згідно з формулою (12):

$$arth(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots = \sum_{n=0}^{\infty} \frac{1}{2n+1} x^{2n+1} \quad |x| < 1 \tag{12}$$

Даний ряд може бути легко обчислений паралельно, оскільки всі його члени незалежні один від одного. Проте, при введенні умови досягнення результатом певної бажаної точності. в паралельному алгоритмі обчислення цього ряду з’являється необхідність постійної комунікації між обчислювачами, викликана необхідністю після кожного обчислення проміжного результату і відправки його ініціатору отримання від нього інформації про досягнення чи недосягнення бажаної точності на поточній ітерації.

Тобто, ініціатор обчислень відповідає за зчитування вхідних даних (x та показник точності в форматі дробового числа), передачу x іншим вузлам, передачі їм позиції члена ряду, який вони обраховують (або номеру поточної ітерації для всіх разом), збір результатів та повідомлення вузлам про продовження чи завершення виконання.

Відповідно, користь цієї задачі для тестування в тому, що зі збільшенням бажаної точності збільшується і кількість міжвузлових комунікацій.

Відносною одиницею навантаження на підзадачу для цього алгоритму є елемент ряду. Проте, зважаючи на невелику кількість вузлів, доцільніше прийняти одиницею навантаження на підзадачу не один елемент ряду, а певну невелику кількість елементів, тобто проміжок ряду.

На рисунку 3.6 зображено графік залежності коефіцієнту прискорення даної програми від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання.

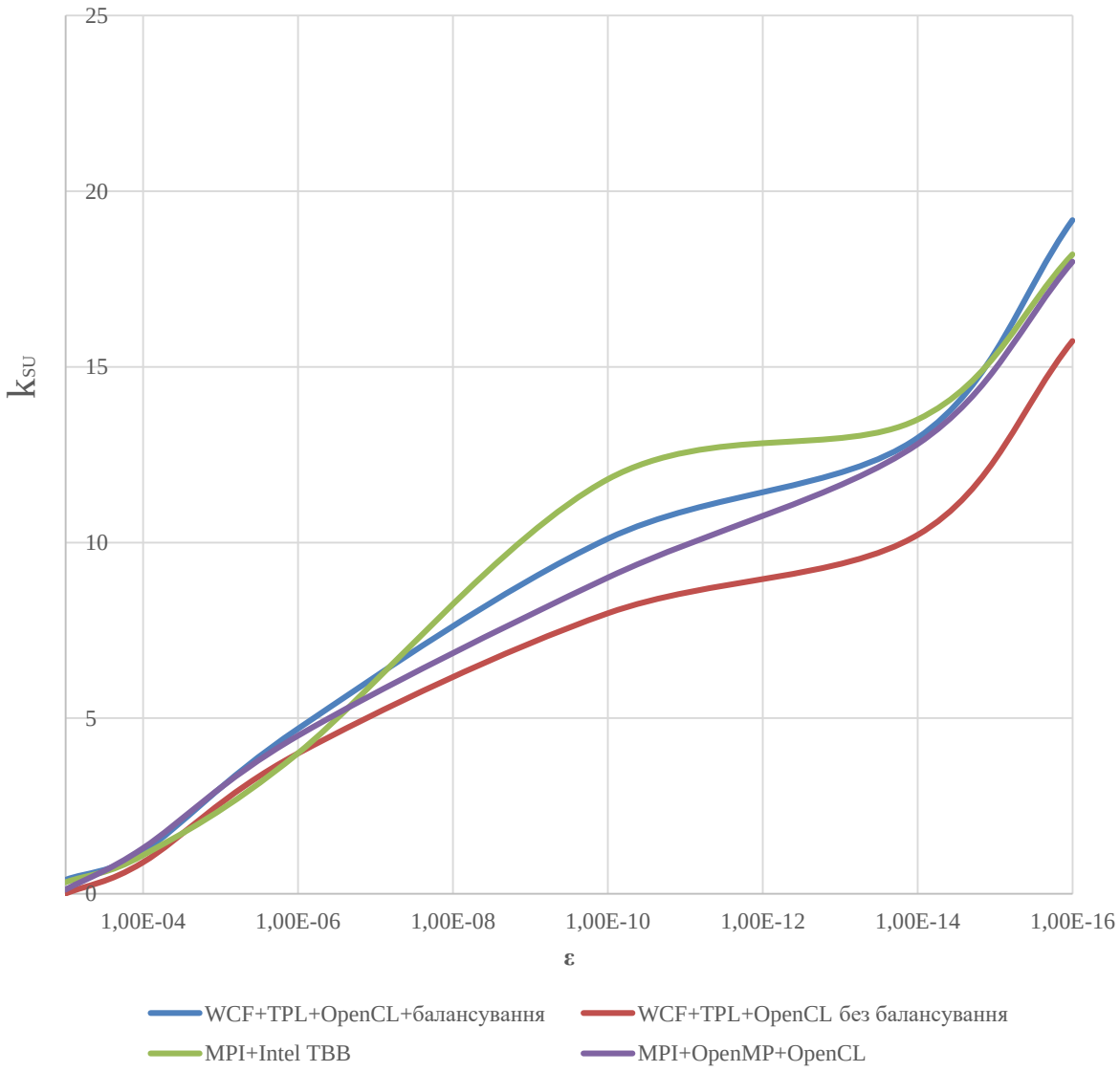

Рис. 3.6. Графік залежності коефіцієнту прискорення програми обрахування наближення ряду від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.7 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання.

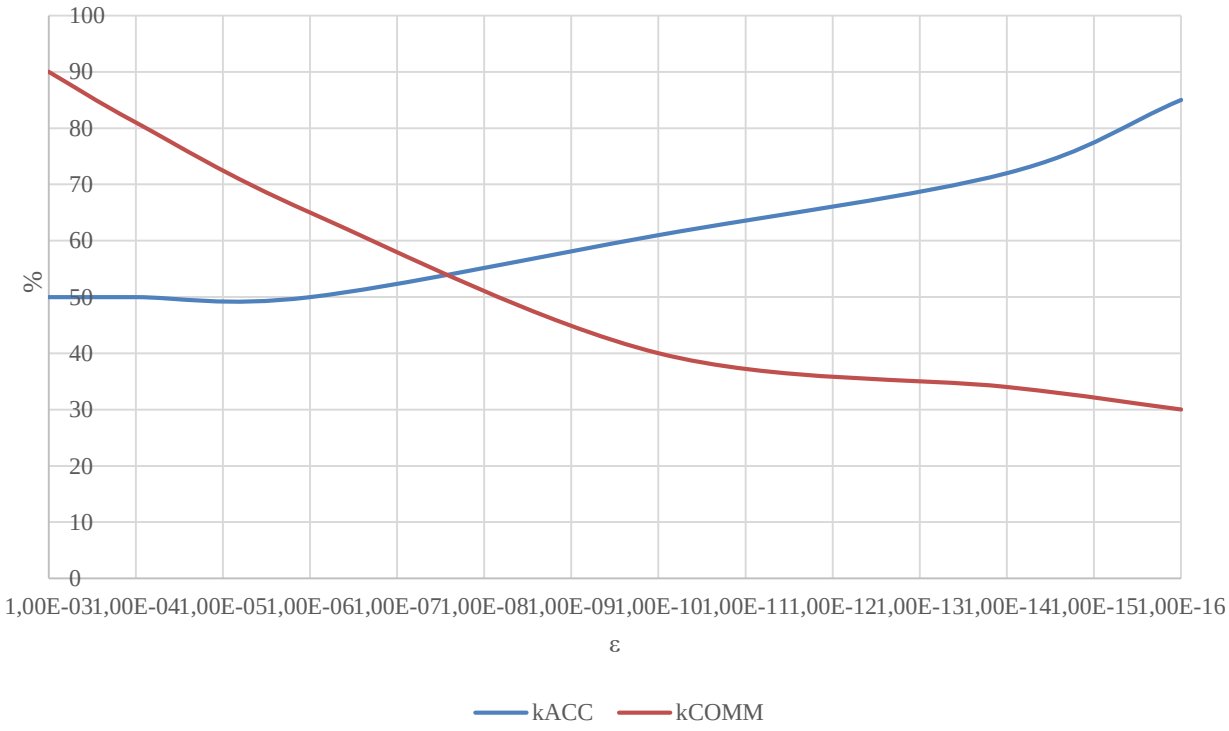

Рис. 3.7. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми обрахування наближення ряду від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання

3.6.3. Обчислення інтегралу методом Сімпсона (парабол)

Метод Сіпсона (парабол) широко застосовується при обчисленні визначеного інтегралу певної підінтегральної функції із заданою точністю. Його ідея полягає в розбитті проміжку інтегрування та обчисленні за формулою (13):

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) \tag{13}$$

Ініціатор обчислень (головний потік або вузол) розсилає вхідні дані іншим обчислювачам. Після цього вони розпочинають обрахування на своїх відрізках. та обраховує їх сумарне значення до тих пір, доки не буде досягнута необхідна точність. Після цього результати відправляються ініціатору. Оскільки кожен обчислювач рахує проміжний результат до досягнення заданої точності, то

можна сказати що і згортка проміжних результатів буде не менше заданої точності. Тому відпадає необхідність ініціатору очікувати отримання проміжних результатів від усіх вузлів. Це зручно реалізувати всіма низькорівневими засобами контрольних прототипів, але в запропонованому методі передбачається що обчислення вважається завершеним тільки після отримання всіх даних. Реалізація зворотного потребує додаткового введення обробки результатів на більш низькому рівні, тому тестування відповідних моделей необхідно провести в двох екземплярах – з очікуванням всього результату та без нього.

Також цінність для тестування цієї задачі полягає в тому, що ініціатор обчислень збирає проміжні результати в хаотичному порядку – тоді немає необхідності почергово очікувати результат від задач. Це відповідає описаній схемі асинхронного надходження результатів від вузлів, тобто є природнім для запропонованого методу.

В цій задачі надзвичайно зручною є можливість поділу кожною задачею свого проміжку на ще менші проміжки, при мінімумі колективних обмінів даними між вузлами, що дозволяє оцінити роботу динамічного балансування між вузлами та в рамках кожного з них.

Відносною одиницею навантаження на підзадачу для цього алгоритму є частина проміжку інтегрування.

На рисунку 3.8 зображено графік залежності коефіцієнту прискорення даної програми від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання.

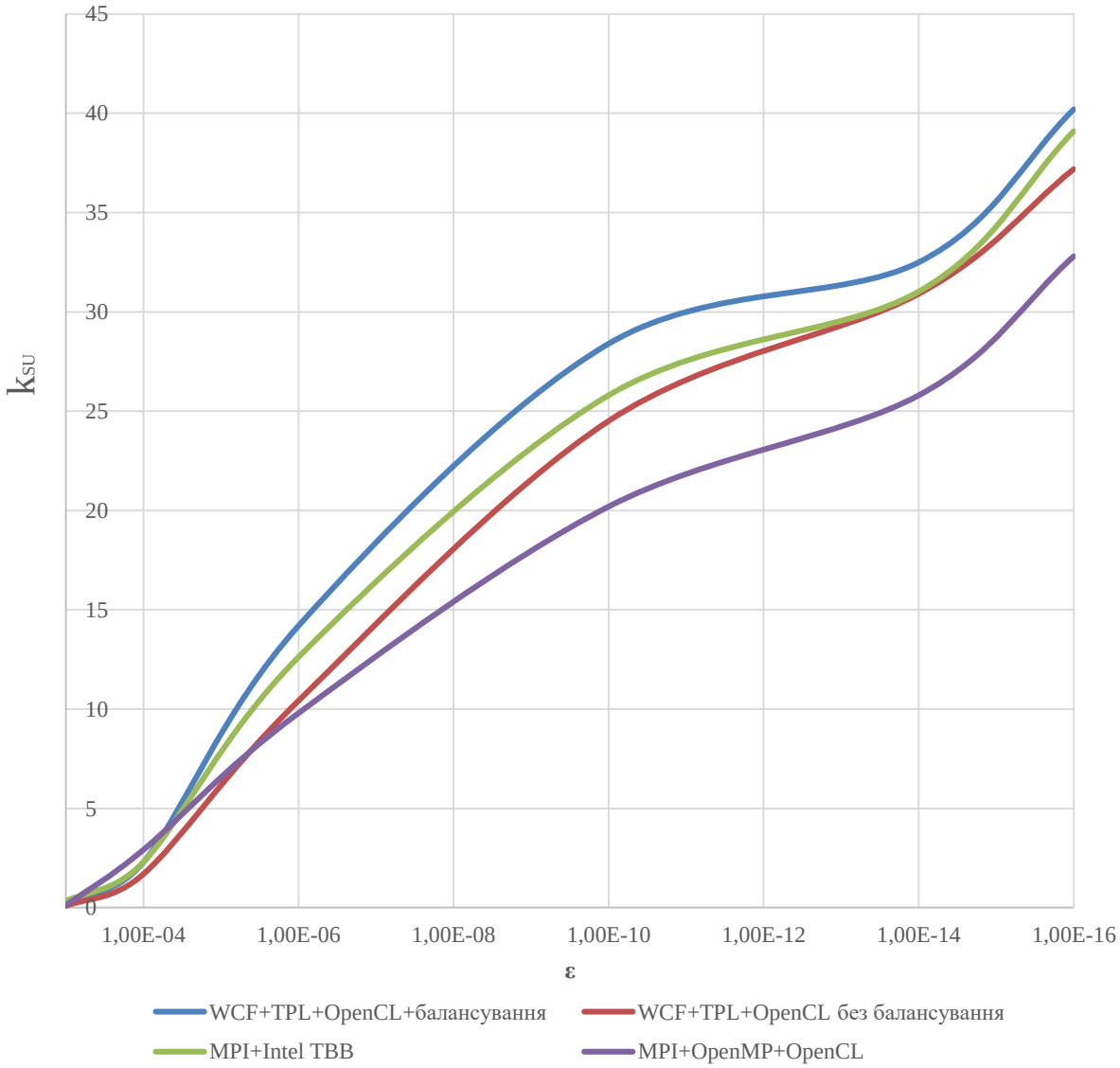

Рис. 3.8. Графік залежності коефіцієнту прискорення програми обрахування визначеного інтегралу функції за методом Сімпсона (парабол) від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.9 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення точності (ε) для кожного програмного комплексу, в якому велося розв’язання.

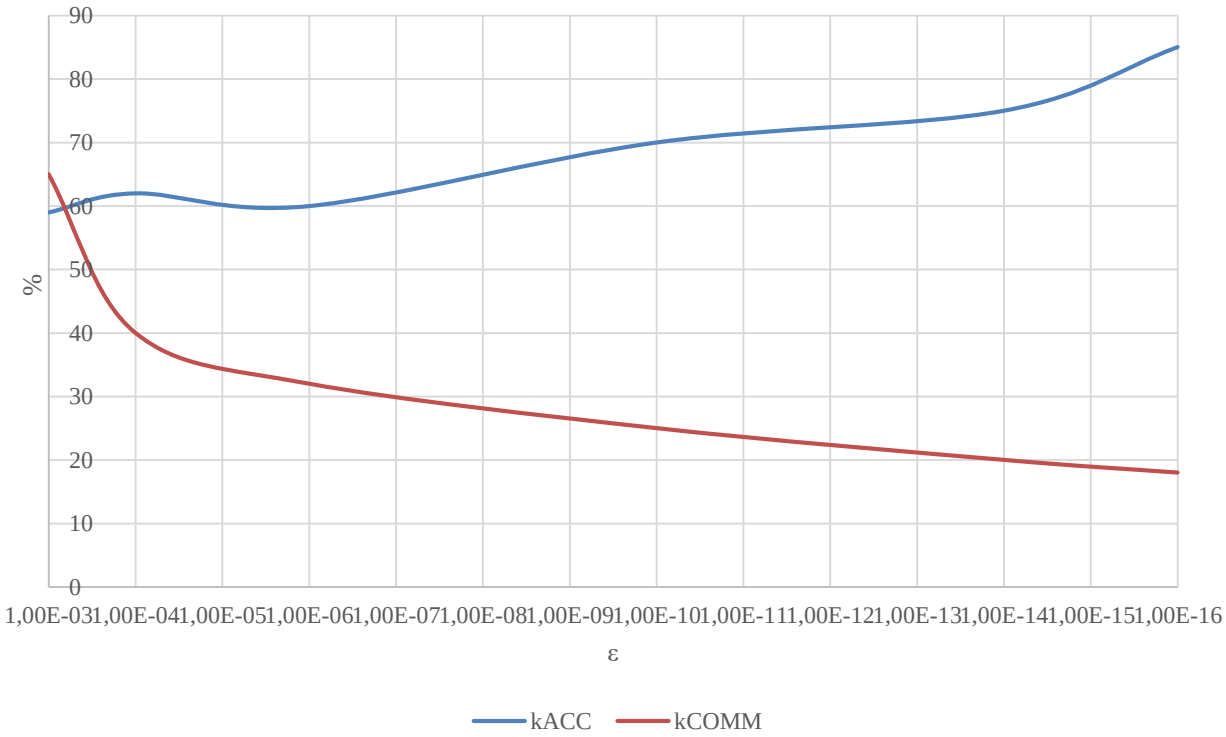

Рис. 3.9. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми обрахування визначеного інтегралу функції за методом Сімпсона (парабол) від значення точності (ε) для кожного програмного комплексу, в якому велося розв'язання

3.6.4. Підрахунок кількості зустрічей кожного слова у документі за моделлю MapReduce

Модель обробки даних MapReduce є однією із ключових в рамках технологій BigData, і призначена для однотипної паралельної обробки великих обсягів даних (зазвичай – для виконання над ними певної операції згортки).

Для алгоритму підрахунку кількості зустрічей кожного слова у документів відповідно до цієї моделі послідовність дій буде наступною:

1. На етапі Map вузол-ініціатор здійснить розбиття документу на частини та розсилку цих частин до вузлів згідно із запропонованим методом;
2. Вузли при прийомі цих частин здійснюватимуть одразу операцію Map, перетворюючи кожне прийняте слово в список пар «ключ-значення», де ключем є слово, а значенням 1;

3. Вузли виконають операцію Reduce, яка полягає у виконанні операції згортки над отриманим списком «ключ-значення» за принципом підрахунку кількості співпадінь кожного ключа. Якщо знайдено співпадіння між двома ключами, то одна відповідна пара «ключ-значення» видаляється зі списку, а в іншої показник «значення» збільшується на 1;
4. Вузли передають ініціатору результуючі списки «ключ-значення», ініціатор проводить остаточну операцію Reduce по всіх отриманих часткових результатах.

Ця задача є типовою прикладною задачею для розподілених комп'ютерних кластерних систем, яка при цьому добре піддається розпаралелюванню, а в контексті запропонованого методу дозволить перевірити ефективність балансування навантаження.

Відносною одиницею навантаження на підзадачу для цього алгоритму є частина документу, тобто масив слів.

На рисунку 3.10 зображено графік залежності коефіцієнту прискорення даної програми від значення розміру документу (кількості слів в ньому N) для кожного програмного комплексу, в якому велося розв'язання.

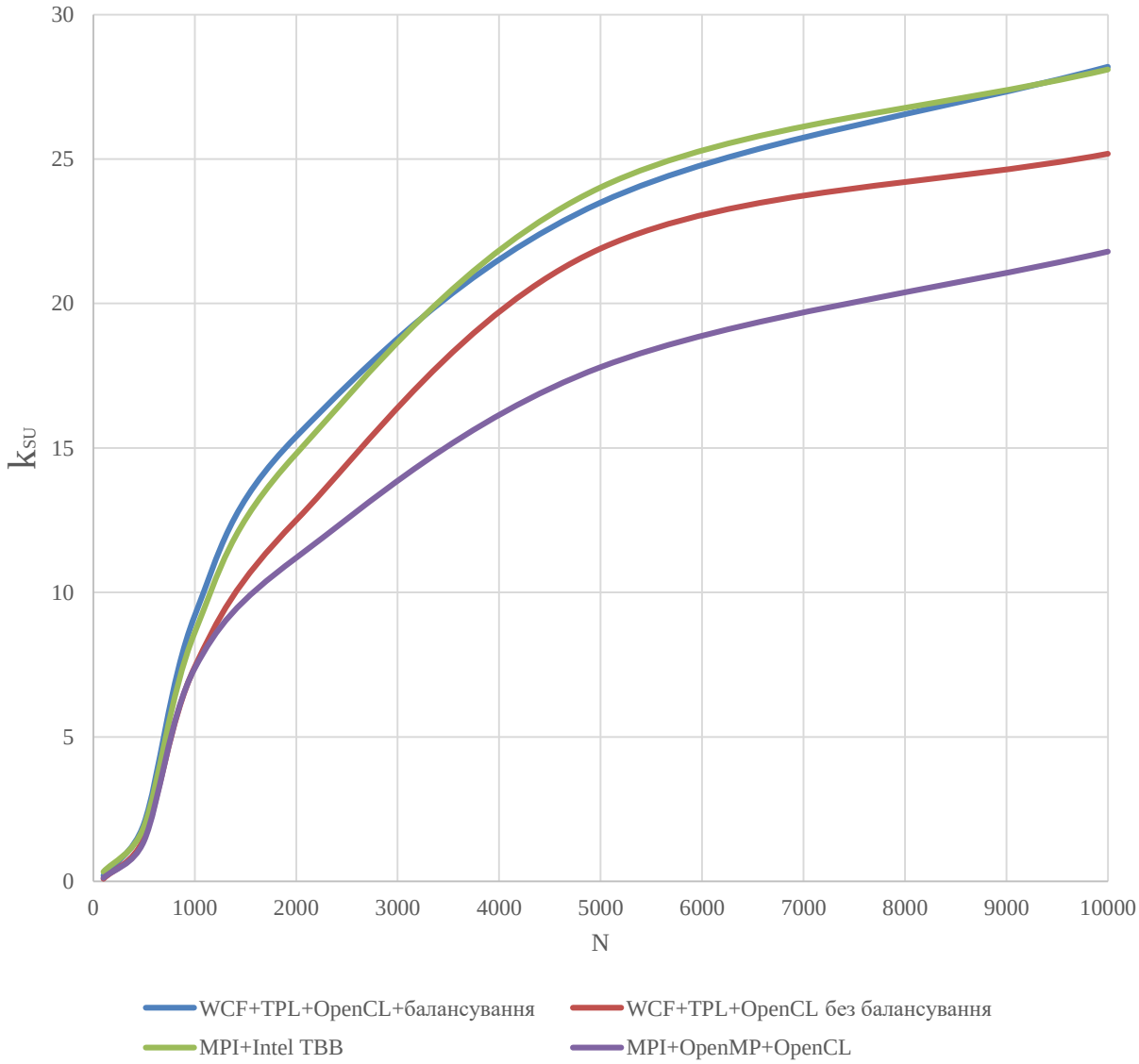

Рис. 3.10. Графік залежності коефіцієнту прискорення програми обрахування кількості зустрічей кожного слова у документі від значення розміру документу (кількості слів в ньому N) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.11 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення розміру документу (кількості слів в ньому N) для кожного програмного комплексу, в якому велося розв’язання.

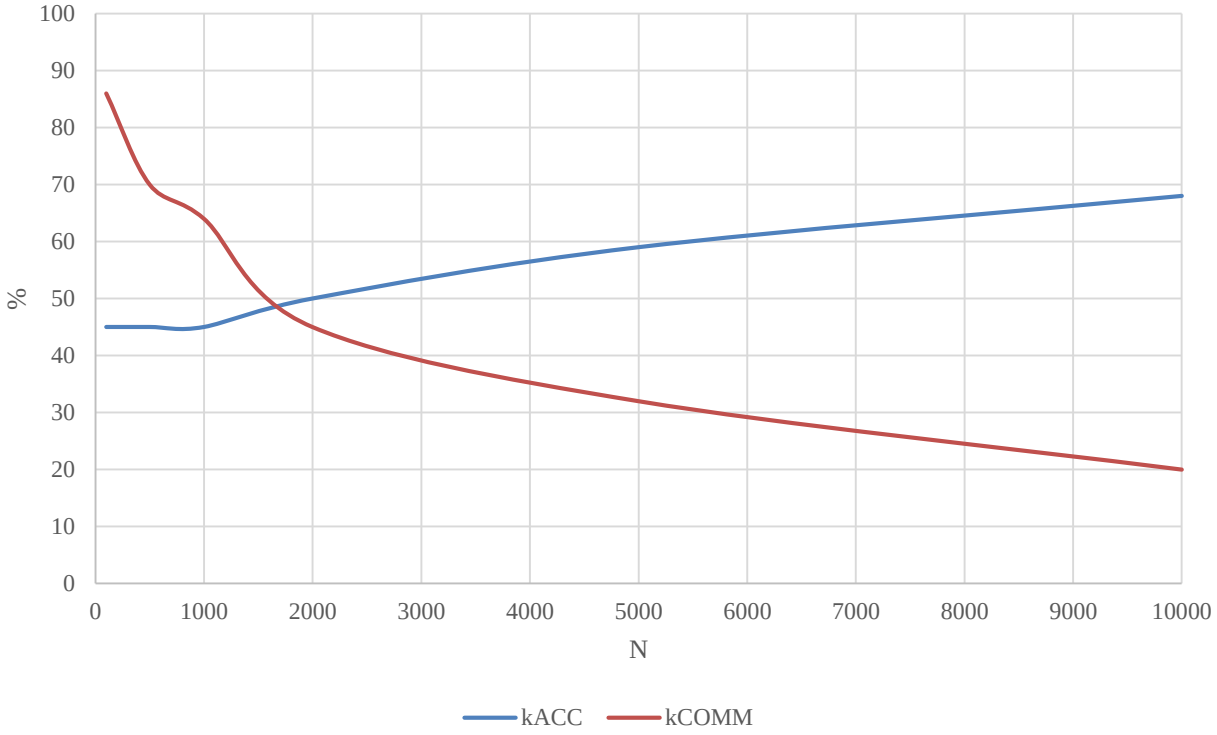

Рис. 3.11. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми обрахування кількості зустрічей кожного слова у документі від значення розміру документу (кількості слів в ньому N) для кожного програмного комплексу, в якому велося розв'язання

3.6.5. Розпізнавання друкованого тексту нейронною мережею

Для тестування в цільових системах буде розгорнуто нейронну мережу, запропоновану в рамках досліджень [13-14].

В загальному виді послідовність дій буде наступна:

1. Вузол-ініціатор отримує зображення, на якому необхідно провести розпізнавання тексту, виконує алгоритм [13] сегментації символів;
2. Вузол-ініціатор розсилає весь алфавіт (набір контрольних символів, який являється пам'яттю нейронної мережі і який сформований внаслідок попередніх запусків та навчання) усім вузлам системи, а також, відповідно до прийнятої моделі розподілу даних розсилає всім вузлам їх частину сегментованих символів до розпізнавання;

3. Кожен вузол приймає набір призначених йому символів та переводить їх у нормовані до чорно-білого спектру вектори пікселів;
4. Далі кожен вузол послідовно проводить розпізнавання кожного символу, шляхом визначення міри його збіжності з кожним із символів алфавіту; При чому сам механізм визначення міри збіжності виконується у внутрішньому паралельному режимі. Відповідно кожен вузол однозначно надає висновок про кожен із символів, які він розпізнавав;
5. Вузол-ініціатор збирає результати розпізнавання у форматі підрядків результуючого текстового рядка та підставляє їх на відповідні їм місця.

Відносною одиницею навантаження на підзадачу для цього алгоритму є набір символів до розпізнавання.

Користь даної задачі для тестування в тому, що в ній від початку присутній механізм внутршнього розпаралелювання функції вирахування міри збіжності, яка є частиною кожної функції, які будуть виконуватись кожним вузлом. Це дозволяє оцінити перспективи необхідності залишати користувачу механізми і для власного ручного розпаралелювання їм в рамках основної функції.

На рисунку 3.12 зображено графік залежності коефіцієнту прискорення даної програми від значення кількості вхідних символів (N) для кожного програмного комплексу, в якому велося розв’язання.

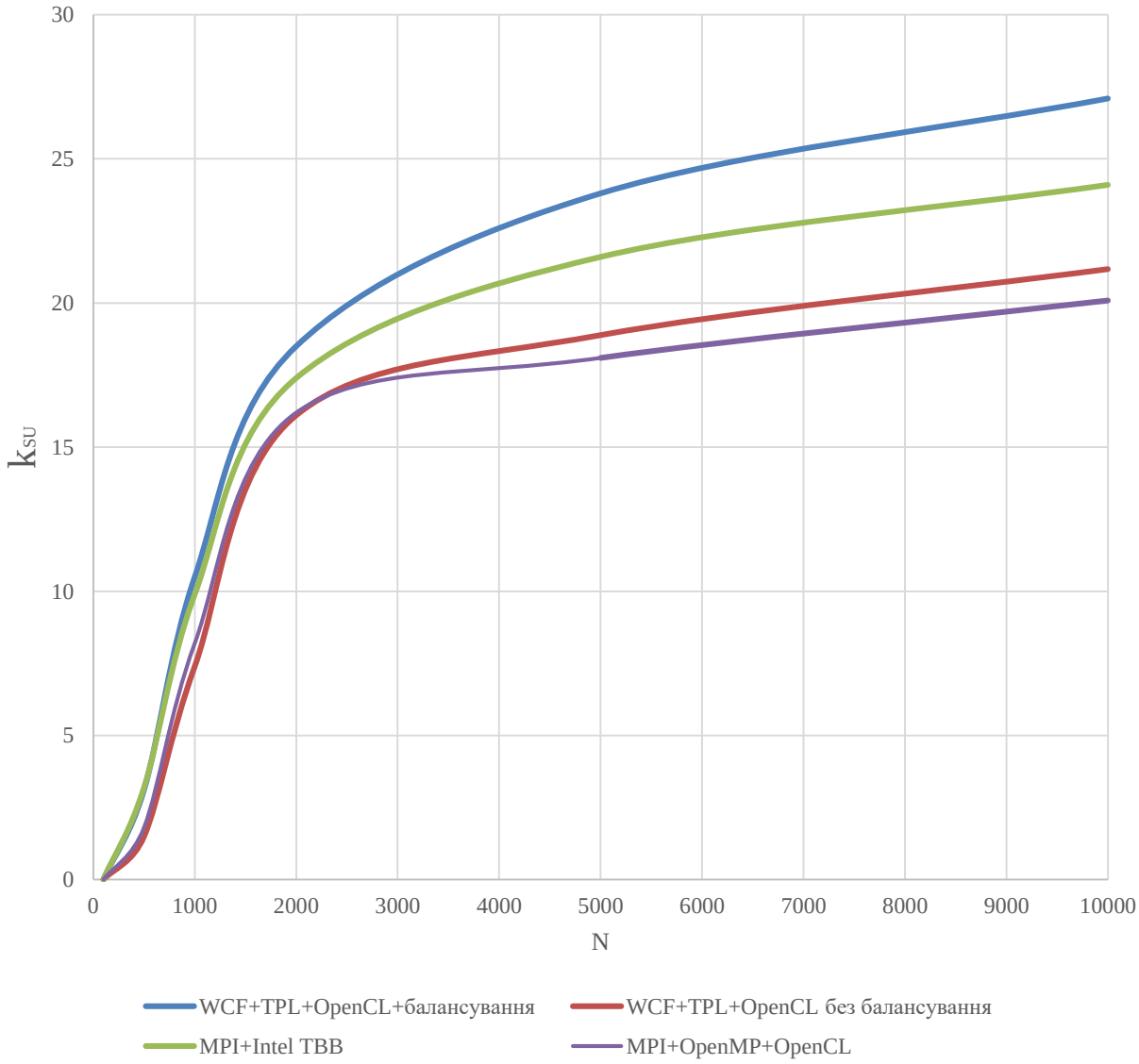

Рис. 3.12. Графік залежності коефіцієнту прискорення програми розпізнавання друкованих символів на основі нейронної мережі від значення кількості вхідних символів (N) для кожного програмного комплексу, в якому велося розв’язання

На рисунку 3.13 зображено графіки залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} даної програми від значення кількості вхідних символів (N) для кожного програмного комплексу, в якому велося розв’язання.

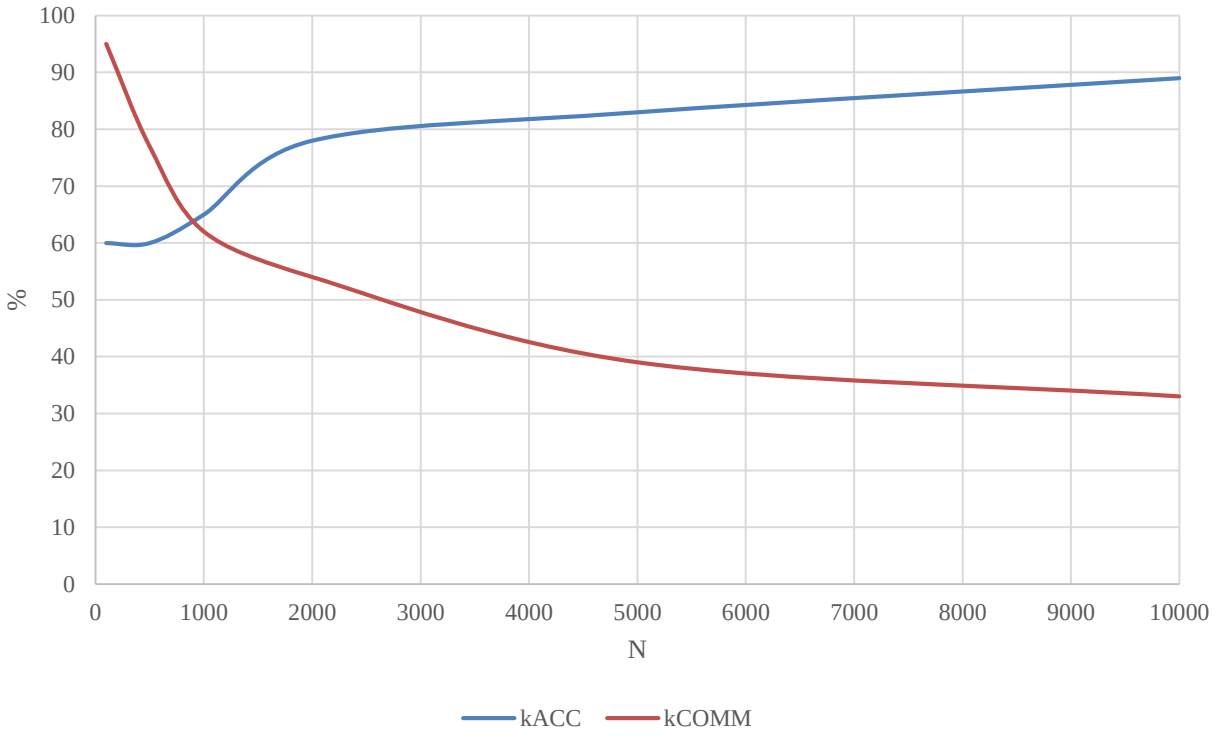

Рис. 3.13. Графік залежності коефіцієнту використання акселераторів k_{ACC} та коефіцієнту використання мережі прискорення k_{COMM} програми розпізнавання друкованих символів на основі нейронної мережі від значення кількості вхідних символів (N) для кожного програмного комплексу, в якому велося розв’язання

3.7. Перевірка розкиду латентності

Важливим фактором, який впливає на продуктивність розподілених систем, комунікативним середовищем який є глобальна мережа, та який може стати чинником принципових помилок на етапі оціни навантаження та проведенні статичного балансування в таких системах є розкид латентності. Наступне тестування для обох розглянутих технологій організації рівня розподіленої системи (WCF та MPI) перевіряє середні показники відхилення за модулем часу передачі відповідних між собою пакетів від найбільшого із отриманих на всіх тестах середнього графіку. На рисунку 3.14 наведено графік, отриманий в результаті цього тестування.

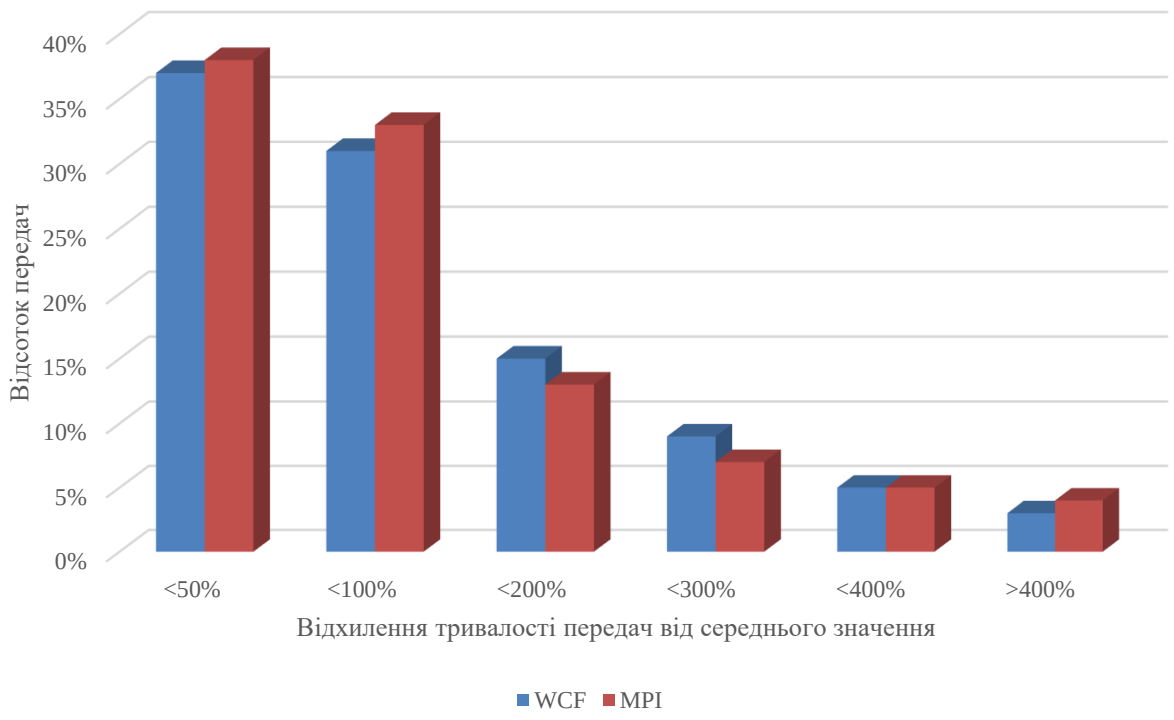

Рис. 3.14. Середні показники відхилення за модулем латентності від найбільш частого (найоптимальнішого) середнього значення

Висновки до розділу 3

В даному розділі описано процес розробки програмного забезпечення для перевірки висунутих ідей і гіпотез, а також проведено тестування розробленого програмного забезпечення.

Було розроблено пакет тестових програмних комплексів для організації роботи розподілених гетерогенних комп'ютерних систем із акселераторами, за які було розгорнуто в двох реальних хмарних РГКС. В рамках кожного із програмних комплексів було організовано розв'язання ряду поширених математичних та прикладних задач, а саме:

1. Множення двох матриць;
2. Проведення векторно-матричних обчислень за заданою формулою;
3. Обчислення наближення розкладу функції в ряд;
4. Обчислення визначеного інтегралу за методом Сімпсона (парабол);
5. Підрахунок кількості входження всіх слів в документ за моделлю MapReduce;
6. Розпізнавання друкованого тексту засобами нейронної мережі.

За результатами обрахунку кожної із цих задач із різними вхідними параметрами можна сформувати ряд висновків:

1. Застосування всіх розглянутих засобів та технологій організації паралельних, розподілених та відвантажених обчислень є ефективним, та приводить до значного зростання коефіцієнтів прискорення програм;
2. Запропонований підхід до організації паралельних обчислень в РГКС із акселераторами на основі застосування технологій WCF, TPL та OpenCL демонструє більшу ефективність, ніж при застосуванні технологій MPI у зв'язці з OpenMP+OpenCL або Intel TBB, але тільки для тих задач, в яких немає частого обміну даними між вузлами в процесі обчислень. Для задач зі значною інтенсивністю комунікацій між вузлами ситуація діаметрально протилежна;

3. Запропонований метод оптимізації паралельних обчислень в РГКС із акселераторами на основі багаторівневого балансування навантаження при його застосуванні в програмних комплексах, які розгортались в системі 1 (сильно неоднорідній) забезпечив додаткове зростання коефіцієнтів прискорення:
   3.1. Порівняно із аналогічним програмним комплексом, але без застосування в ньому балансування в середньому на 17%, при цьому максимальне зростання на 22% спостерігається для задач із незначними комунікаціями в процесі роботи, і мінімальне зростання на 13% для задач зі значною інтенсивністю комунікацій в процесі роботи;
   3.2. Порівняно із програмним комплексом, в якому застосовувались MPI та Intel TBB і лише внутрішні методи балансування цих технологій та балансування операційних систем вузлів середнє зростання коефіцієнту прискорення склало 7%, при цьому максимальне зростання на 10% спостерігалося для задачі множення матриць, а мінімальне зростання на 3% для задачі MapReduce;
   3.3. Порівняно із програмним комплексом, в якому застосовувались MPI, OpenMP та OpenCL і лише внутрішні методи балансування цих технологій та балансування на рівні операаційних систем вузлів середнє зростання коефіцієнту прискорення склало 18%, при цьому максимальне зростання на 20-24% спостерігалося для задач із незначними комунікаціями в процесі роботи, а мінімальне зростання на 5% для задач зі значною інтенсивністю комунікацій в процесі роботи;
4. В усіх випадках максимальних коефіцієнтів прискорення вдалося досягти при організації їх вирішення засобами WCF, TPL та розробленого методу балансування навантаження, але варто зауважити, що порівняно із технологіями Intel TBB зростання є невеликим, не зважаючи на те, що комунікація на рівні розподіленої системи для Intel TBB організаована

засобами MPI, що може свідчити про більш ефективну підтримку акселераторів та паралелізму на рівні вузлів в Intel TBB, ніж в OpenCL та TPL;

5. Для всіх задач зі зростанням об’ємності обчислень, які необхідно провести для їх вирішення, зростає і роль акселераторів в їх виконанні (в рамках запропонованого методу балансування), не залежно від інтенсивності комунікацій (що можна пояснити тим, що за принципом функціонування системи кількість міжвузлових передач (по повільніших мережевих каналах зв’язку) максимально зменшується, а от кількість передач через інтерфейс зв’язку з акселератором (який значно продуктивніший за мережеві канали) не підлягає зменшенню). При незначних об’ємах обчислень на акселераторах виконувалось близько 50% обчислень, але при зростанні об’ємів обчислень спостерігалось і значне зростання ролі акселераторів, сягаючи до 90%;
6. В той же час зі зростанням об’ємності обчислень спостерігається падіння середнього відсотку сумарного часу виконання задачі, елементи в тій чи іншій мірі простоювали внаслідок обміну даними. При невеликих обсягах обчислень в середньому 90% часу роботи програми займали комунікації, проте завжди цей показник досить стрімко падав, і, в залежності від природнього рівня комунікацій, який вимагається самим алгоритмом вирішення задачі, становив від 13-18% (для задач множення матриць та інтегрування) до 30-33% (для задач обчислення ряду та розпізнавання тексту);
7. Варто також відмітити, що при найменших обсягах необхідних для вирішення задачі обчислень майже для всіх задач коефіцієнт прискорення був менше 1, що свідчить про те, що застосування РГКС для вирішення невеликих задач є неефективним підходом. При тому і для середніх за обсягом обчислень приріст коефіцієнтів прискорення був доволі незначним, можна допустити, що схожих коефіцієнтів прискорення можна

було б досягнути при вирішенні задач лише в рамках паралельної, а не розподіленої системи;

8. Середні показники відхилення за модулем латентності як для MPI, так і для WCF були в межах норми для глобальних мереж і відповідають половинному нормального розподілу, але при цьому із графіку 3.14 слідує, що для WCF в загальному відхилення були трохи меншими, ніж для MPI;
9. Необхідно також відмітити, що додаткове тестування, яке було проведено в системі 2 (контрольній) показало, що у випадку, коли всі вузли системи однакові, то виконання запропонованого методу балансування не несе таких значних приростів коефіцієнту прискорення, а навіть навпаки – операції, виконання яких вимагається для виконання балансування, дещо сповільнюють роботу програми.

ВИСНОВКИ

В ході виконання першого етапу магістерської дисертації було проведено аналіз сучасних підходів до організації паралельних програм, основних моделей їх організації, основних технологій, які втілюють працюють за цими моделями. Було виявлено, що сучасні паралельні комп'ютерні системи не можуть забезпечити вирішення сучасних наукоємних задач за прийнятні проміжки часу, тому найбільш застосовуваними та найперспективнішими комп'ютерними системами для обробки високонавантажених обчислень є розподілені комп'ютерні системи. При тому все більшої популярності набирають не кластерні (гомогенні, повнозв'язні), а гетерогенні розподілені комп'ютерні системи, без вимог повної зв'язності. А найбільш новітні підходи та технології організації обчислень в РГКС дозволяють також розподілити в них обчислення і на акселератори (якими найчастіше являються графічні процесори), якщо такі наявні на вузлах.

В той же час жодна із сучасних принципових схем обробки завдань в розподілених комп'ютерних системах, не залежно від того, які технології використовуються не включає в себе на жодному з рівні етапу динамічного балансування. Більше того, навіть початковий розподіл навантаження (статичне балансування) при організації в рамках розглянутих найпопулярніших технологіях не враховує гетерогенність складових елементів, ця задача повністю покладається на розробника. Також сучасні технології для РГКС будуються навколо низькорівневих моделей паралельного програмування, які хоч і забезпечують більшу гнучкість та швидкодію, проте мають ряд суттєвих недоліків порівняно з високорівневими моделями, а саме – відсутність автоматичної регуляції зерна паралелізму під час роботи, наближення створюваної програми до особливостей системи, а не особливостей задачі, можливість виникнення помилок на системному рівні через неправильне застосування низькорівневих примітивів, а, відповідно, вимагання високого рівня кваліфікації розробників паралельних програм для таких систем. Більш

детально висновки, зроблені за результатами проведеного аналізу, наведено у розділі висновків до розділу 1.

Виходячи з проведеного аналізу, в другому розділі даної магістерської дисертації було запропоновано метод оптимізації паралельних обчислень в РГКС з підтримкою акселераторів, на основі багаторівневого балансування навантаження. Було виділено новий рівень абстракції на схемі організації обробки завдань в РГКС, що дозволило проводити ефективне та природнє балансування як в рамках одного вузла, так і між вузлами. Крім того, метод базується на абстракціях високого рівня (проте і з підтримкою деяких низькорівневих абстракцій), що дозволяє втілити всі переваги таких абстракцій, описані вище.

Статичне балансування в РГКС було запропоновано виконувати на основі комплексної оцінки цільової РГКС в контексті виконання даної задачі, для чого було запропоновано метод оцінки продуктивності елементів системи та каналів зв’язку між ними на основі як їх фізичних показників, так і на основі непрямих показників рахункових особливостей задачі. Саме оцінювання задачі було запропоновано проводити на основі побудови та аналізу її абстрактного синтаксичного дерева. Таке балансування є достатньо наближеним, але водночас не потребує високих затрат часу. Статичне балансування виконується перш за все на рівні розподіленої системи, але передбачає планування аж до конкретних елементів вузлів, тобто захоплює частину рівня паралельної системи.

Наближення, а отже деяку неоптимальність отриманого статичним балансувальником розподілу пропонується компенсувати введенням динамічного балансування, для чого запропоновано підхід до оцінювання комплексних фізичних показників кожного вузла та елементу системи. Виконується воно перш за все на рівнях паралельних систем, проте з можливістю міграції підзадач між вузлами системи, тобто захоплює частину рівня розподіленої системи.

Для обробки ключової задачі міграції процесів між вузлами, без чого не можливе повноцінне динамічне балансування, було запропоновано альтернативний підхід, який базувався на початковому значному розподіленні задачі та оперуванні не процесами, а отриманими підзадачами підзадач.

З метою додаткової оптимізації, використовувались моделі не лише паралельного, а й асинхронного програмування (відкладені обчислення, відкладені передачі даних).

На загальному рівні були розглянуті питання убезпечення системи від стресових ситуацій (обриву зв’язку між вузлами) та потенційних атак.

На основі запропонованого методу в третьому розділі магістерської дисертації було проведено розробку пакету програмних комплексів для організації процесу паралельної обробки задач в РГКС з акселераторами. Програмні комплекси були розроблені з використанням оглянутих в першому розділі найпопулярніших технологій та із застосуванням запропонованих альтернативних технологій, при тому задля прозорого тестування деякі комплекси включали запропонований метод, а деякі – ні.

Також в третьому розділі було проведено тестування всіх розроблених комплексів при розгортанні їх в двох реальних РГКС із акселераторами, організованих в віртуальному хмарному середовищі Google Cloud Console. Тестування проводилося в контексті організації обчислення в рамках розроблених комплексів ряду прикладних поширених задач, а саме виконання векторно-матричних операцій, обчислення наближення розкладу функції в ряд, обчислення визначеного інтегралу, обробка об’ємних текстових документів, розпізнавання друкованого тексту засобами нейронних мереж.

Тестування показало значні можливості до скорочення часу роботи програми при організації паралельних обчислень в РГКС всіма розглянутими засобами, коефіцієнти прискорення для найбільш природньо розпаралелюваних задач сягали в середньому близько 30. Застосування запропонованого методу оптимізації на основі балансування та супутніх підходів додатково дозволило збільшити коефіцієнти прискорення для в залежності від задачі на 6-20%,

порівняно із програмами, в яких це балансування не проводилось. Зазначимо, що цей приріст вищий, ніж отриманий професором Стіренко в дослідженні [17], що пояснюється тим, що тестова система має акселератори та значний ступінь відмінності між всіма акселераторами та всіма процесорами, що і створює заділ для більшого виграшу від балансування, ніж в однорідних кластерних системах, на яких проводилось тестування в дослідженні [17]. На користь цього факту свідчить також те, що застосування описаного методу в гомогенних системах (див. підрозділ 3.4, тестова система 2 – контрольна) приводило до менших показників прискорення, ніж у випадку, коли метод взагалі в них не застосовувався. Більш детально висновки, зроблені за результатами проведеного тестування, наведено у розділі висновків до розділу 3.

Недоліки, зауваження та пропозиції їх вирішення у подальшій роботі

Основним недоліком розробленого методу можна вважати його орієнтованість на регулярний та умовний нерегулярний паралелізм за паттерном «Data-driven parallelism». Для його вирішення необхідно на етапі оцінки задачі не обмежуватись лише поверхневим аналізом абстрактного синтаксичного дерева задачі, а проводити глибокий аналіз, для виявлення всіх можливих підзадач та зв’язків між ними, і за результатами цього будувати паралельний граф кожної задачі, на основі якого вже і виконувати подальше планування та балансування навантаження. Відкритим лишається питання оптимізації затрат часу для проведення такого аналізу.

З переходом до планування обчислень та балансування навантаження на основі паралельного графу задач можна застосувати як стандартні евристичні алгоритми для гетерогенних систем, такі як модифікований для підтримки акселераторів HEFT (англ. Heterogeneous Earliest Time First) [31], або модифікувати запропонований авторами дослідження [32] алгоритм на основі застосування методу гілок і меж (англ. Branch-and-Bound) із пошуком початкового для нього наближення оптимального розв’язку для нього за алгоритмом імітації відпалу (англ. Simulated annealing) рішення як основи для нього.

Перспективною може бути розробка методу початкових тестових запусків паралельних задач із невеликими вхідними даними, з метою більш точнішого зважування карти-графу РГКС, або ж, як альтернатива, зважування можна проводити за результатами попередніх запусків програм.

Ще одним важливим недоліком є те, що метод передбачає, хай і оптимізоване, але завантаження всіх вузлів системи, навіть при їх мінімальній продуктивності. Як показало тестування, при виконанні за такою схемою невеликих задач, коефіцієнт прискорення стабільно був нижче 1, що свідчить про те, що розпаралелювання на всю РГКС було менш оптимальним, ніж якби проводилось на лише одному вузлі або групі окремих вузлів. Тому необхідно вдосконалити процес перш за все статичного балансування, аби він враховував можливість наявності оптимальнішого вирішення задачі в разі застосування не всіх вузлів системи для обчислень. Це створює також перспективи для організації не почергового, а одночасного виконання в системі декількох різних задач, тобто перехід до схеми роботи «система масового обслуговування». Початковий заділ на це в запропонованому методі передбачався, для цього і було запропоновано використовувати не пряму делегацію виконання різних функцій різним вузлам, а упаковку цих функцій в динамічні об’єкти єдиного класу, що, при введенні механізму індексації цих об’єктів не лише в рамках підзадач, а й в рамках глобальної задачі, дозволить організувати єдині черги виконання таких об’єктів на кожному з вузлів, не залежно від самого змісту функцій, упакованих в ці об’єкти.

Також необхідно розширити спектр підтримуваних акселераторів, особливо зосередившись на підтримці TPU та FPGA. Наприклад дослідження [33] показує, що для розглянутих в ньому задач використання як окремо FPGA, так і зв’язок FPGA з CPU та/або GPU забезпечує найзначніші зростання коефіцієнтів прискорення. Тому можна допустити, що перспективним може бути розгортання РГКС, в якій кожен вузол міститиме FPGA, а центральним процесорам відводитиметься лише роль загального менеджменту та забезпечення зручного інтерфейсу взаємодії з FPGA. Додатково, при організації

ефективного засобу програмування мовою високого рівня загального призначення обчислень в FPGA (з внутрішнім переведенням в мову VHDL або Verilog) можна організувати розподілену систему на базі FPGA. Подібні технології, які забезпечують можливості такого програмування вже маються, наприклад Hastlayer для мови C#, Catapult C, Cynthesizer, HDL coder від компанії Mathworks, HLS compiler від компанії Intel для мови C++, Reconfigure.io для мови Golang.

Бажаним є проведення більш детального та глибокого тестування методу після введення до нього всіх запропонованих покращень, в тому числі на реальних, а не хмарних РГКС, а також таких, які б включали значно більшу кількість вузлів, ніж було розглянуто в цій роботі.

Ще одним важливим питання безпеки для розглянутого методу є злісні перевантаження. Взагалі від звичайних перевантажень система захищена від самого початку, тому що декларується рівномірний розподіл навантаження з оцінкою обмежень, наданих користувачами, які надали додали свої машини в рамках вузлів. Проте від злісного перевантаження, коли спеціально ведеться захоплення системи, шляхом постійного ініціювання безкінечних обчислень, систему можна захистити, ввівши економічну складову, тобто монетизацію. Передбачити оплату ініціатором обчислень робочого часу, який використають його обчислення на кожному конкретному вузлі власникові цього вузла. Проте це потребує введення додаткової сторони в роботу системи та механізму умовного депонування (ескроу), який передбачає оплату тільки по факту виконання обов’язків обома сторонами договору (в цьому випадку – ініціатором обчислень та кожним окремим власником конкретного вузла системи).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software / H. Sutter // Dr. Dobby's Journal. – 2005. –Vol. 30, no. 3.
2. Martonosi, Margaret. Parallelism, heterogeneity, communication: Emerging challenges for performance analysis / Margaret Martonosi // Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on IEEE. – Washington, DC, USA: IEEE Computer Society, 2012. – P. 124.
3. Software as a service for data scientists / Bryce Allen, John Bresnahan, Lisa Chilers [et al.] // Commun. ACM. – 2012. – Vol. 55, no. 2. – P. 81-88.
4. Tanenbaum, Andrew S. Distributed Systems: Principles and Paradigms (2nd Edition) / Andrew S. Tanenbaum, Maarten van Steen. – Upper Saddle River, NJ, USA: Prentice-Hall, Inc. – 2006.
5. Kim, H. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU) [Електронний ресурс] / H. Kim, R. Vuduc, S. Baghsorkhi // Synthesis lectures on computer architecture. –2012. – Vol. 7, no.2. – P. 1-96. Режим доступу до ресурсу: doi: 10.2200/ S00451ED1V01Y201209CAC020 (дата звернення 01.05.2020).
6. Cheng, L. Intelligent scheduling for simultaneous CPU-GPU applications : M.Sc. in Computer Sciences thesis [Електронний ресурс] / L. Cheng: Graduate College, University of Illinois at Urbana-Champaign, Illinois, USA. – 2017. – Режим доступу до ресурсу: http://rsim.cs.uiuc.edu/Pubs/Lin_thesis.pdf (дата звернення 01.05.2020).
7. Lee, V. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU / V. Lee [et al.] // ACM SIGARCH Computer Architecture News. –2010. – Vol. 38, no. 3. –P. 451-460. doi: 10.1145/1816038.1816021.
8. CPU vs. GPU - Performance comparison for the Gram-Schmidt algorithm [Електронний ресурс] / T. Brandes, A. Arnold, T. Soddemann, D. Reith // The

European Physical Journal Special Topics. – 2012. – Vol. 210. –P. 73–88. – Режим доступу до ресурсу: doi: 10.1140/epjst/e2012-01638-7. (дата звернення 01.05.2020).

9. Performance Evaluation of CPU-GPU with CUDA Architecture Hybrid Computing [Електронний ресурс] / H. Haneesha, B. Chandrashekhar, H. Lakshmi, Sunil // R&D. International Conference on Communication and Computing (ICC-2015), Bangalore. – 2015. – Режим доступу до ресурсу: https://www.researchgate.net/publication/314220596_Performance_Evaluation_of_CPU-GPU_with_CUDA_Architecture (дата звернення 01.05.2020).
10. Performance Characterization of State-Of-The-Art Deep Learning Workloads on an IBM "Minsky" Platform [Електронний ресурс] / M. Guignard, M. Schild, C. Bederián, N. Wolovick // Hawaii International Conference on System Sciences, Hawaii, USA. – 2018. – Режим доступу до ресурсу: doi: 10.24251/HICSS.2018.702 (дата звернення 01.05.2020).
11. Seppälä, S. Performance of Neural Network Image Classification on Mobile CPU and GPU : M.Sc. in Computer Sciences thesis [Електронний ресурс] / S. Seppälä: Aalto University, Espoo, Finland. – 2018. – Режим доступу до ресурсу: https://pdfs.semanticscholar.org/946d/3f843ea93f22cc9c7e30af42a682139ad1e6.pdf. (дата звернення 01.05.2020).
12. Heterogeneous CPU+GPU computing [Електронний ресурс] / A. Varbanescu // ASCI springschool of Heterogeneous Computing Systems, Soesterberg, The Netherlands. –2017. – Режим доступу до ресурсу: https://www.ntnu.edu/documents/139931/1275097249/NTNU_HetComp_toPublish.pdf/486588ee-23af-4104-8a04-bb18cd5a68c1 (дата звернення 01.05.2020).
13. Тизунь Віталій Юрійович. Система розпізнавання тексту на основі гетерогенних комп'ютерних систем : дис. маг. техн. наук / Тизунь Віталій Юрійович. –Київ, 2019. – 89 с. – Бібліогр. : с. 85–88.

14. The Organization of Parallel Computations in Heterogeneous Computing Systems [Електронний ресурс] / V. Demchyk, V. Tyzun, O. Rusanova, A. Korochkin // International Conference ICSFTI2019, Kyiv, Ukraine. –2019. –P. 220–229. – Режим доступу до ресурсу: http://comsys.kpi.ua/katalog/files/proceedings-icsfti-2019-1.pdf (дата звернення 01.05.2020).
15. Дослідження ефективності дрібнозернистого паралелізму в багатоядерних комп'ютерних системах. / В.В. Демчик, О.В. Корочкін, О.В. Русанова // Вісник НТУУ «КПІ». Інформатика, управління та обчислювальна техніка. – 2020. – № 66. – С. 56-61.
16. Застосування дрібнозернистого паралелізму для підвищення ефективності паралельних та розподілених обчислень [Електронний ресурс] / В.В. Демчик, О.В. Корочкін // Міжнародна науково-технічна конференція "Безпека, відмовостійкість, інтелект" (ICSFTI2018), Київ, Україна. – 2018, С. 362 – 368. – Режим доступу до ресурсу: http://comsys.kpi.ua/upload/GN_330-336_DemchykValerii.pdf (дата звернення 01.05.2020).
17. Стіренко С. Г. Організації паралельний обчислювальних процесів в кластерних системах [Текст] / С.Г. Стіренко, – К.: «Три К», 2014. – 196 с.
18. Herlihy, Maurice. The Art of Multiprocessor Programming / Maurice Herlihy, Nir Shavit. – San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, – ISBN:0123705916, 9780123705914.
19. Parsons, R. A++/P++ array classes for architecture independent finite difference computations / R. Parsons, D. Quinlan // Proceedings of the Second Annual Object-Oriented Numeric Conference (OON-SKI'94). – Sun River, OR, USA: IEEE Computer Society, 1994.
20. Представление задач в системах распараллеливания с изменяемой зернистостью / Г. М. Луцкий, С. Г. Стиренко, А. И. Зиненко, Д. В. Грибенко // Вісник НТУУ «КПІ». Інформатика, управління та обчислювальна техніка. – 2012. – № 57. – С. 101-109.

21. Pheatt, Chuck. Intel ® Threading Building Blocks / Chuck Pheatt // Journal of Computing Sciences if Collages. – 2008. – Vol. 23, no. 4. – P. 298.
22. Bataev, A. Towards OpenMP support in LLVM / A. Bataev, A. Bokhanko // EuroLLVM 2013 European LLVM conference. – Paris, France: ENS, 2013.
23. Novillo, D. OpenMP and automatic parallelization in GCC / D. Novillo // Proceedings of the GCC developers' summit. – San Francisco, CA, USA ACM, 2006. – P. 132-141.
24. Marowka, Ami. Performance of OpenMP Benchmarks on Multicore Processors/ Ami Marowka // Proccedings of the Sth international conference on Algorithms and Architectures for Parallel Processing – ICA3PP ’08. – Berlin, Heidelberg: Springer-Verlag. 2008. – P. 208-219.
25. Aiken, A. Optimal loop parallelization / A. Aiken, A. Nicolau // SIGPLAN Not. –1988. – Vol. 23, no. 7. – P. 308-317.
26. ISO/IEC. C++ 2011 Standard Document 14882:2011. – ISO/IEC Standard Group for Information Technology / Programming Languages / C++. – 2011.
27. A Java Fork/Join Framework [Електронний ресурс] / Lea Doug // Proceedings of ACM Java Grande 2000 Conference – San Francisco, CA, USA. – 2000. – Режим доступу до ресурсу: http://gee.cs.oswego.edu/dl/papers/fj.pdf (дата звернення 01.05.2020).
28. Parallel Program Performance Prediction Using Deterministic Task Graph Analysis [Електронний ресурс] / Vikram S. Adve, Mary K. Vernon // ACM Transactions on Computer Systems. – 2004. – Режим доступу до ресурсу: doi:10.1145/966785.966788 (дата звернення 01.05.2020).
29. Brainerd, W.S., Landweber, L.H. Theory of Computation / W.S. Brainerd, L.H. Landweber. — Wiley, 1974.
30. Жуков І.А., Корочкін О.В. Паралельні та розподілені обчислення: Навч. посібник [Текст] / Жуков І.А., Корочкін О.В. // – К.: Корнійчук, 2005. – 226 с. – ISBN 996-7599-36-1.
31. Optimization of the HEFT algorithm for a CPU-GPU environment [Електронний ресурс] / K. R. Shetti, S. A. Fahmy, T. Bretschneider //

International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2013), Taipei, Taiwan. – 2013. – Режим доступу до ресурсу: doi: 10.1109/PDCAT.2013.40 (дата звернення 01.05.2020).

32. Two Phase Algorithm for Load Balancing in Heterogeneous Distributed Systems [Електронний ресурс] / G. Attiya, Y. Hamam // 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing. – 2004. – Режим доступу до ресурсу: doi:10.1109/EMPDP.2004.1271476 (дата звернення 01.05.2020).
33. Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs [Електронний ресурс] / Hamish J. Macintosh, Jasmine E. Banks, Neil A. Kelson // Hindawi International Journal of Reconfigurable Computing. – 2019. – Vol. 2019. – P.13. – Режим доступу до ресурсу: doi:10.1155/2019/3679839 (дата звернення 01.05.2020).